INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Compiling ComLisp to Executable Machine Code: Compiler Construction

Wolfgang Goerigk, Ulrich Hoffmann

Bericht Nr. 9812
Oktober 1998

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Preußerstraße 1-9, D-24105 Kiel
Germany

Compiling Comlisp to Executable Machine Code: Compiler Construction

Wolfgang Goerigk, Ulrich Hoffmann

wg@informatik.uni-kiel.de
uho@informatik.uni-kiel.de

Report No. 9812
October, 1998

Dieser Bericht ist als persönliche Mitteilung zu verstehen.

Abstract

This report is one part of a series of documents describing the fully verified specification, construction and implementation of a ComLisp–compiler. ComLisp is a subset of ANSI-CommonLisp. Programs are systems of first order mutually recursive function and procedure definitions on dynamic Lisp data. ComLisp is both compiler source and implementation language. The work is part of the DFG research project *Verifix* on *Correct Compilers*. The major goal in *Verifix* is to develop methods for correct realistic compiler construction for practically relevant source languages and concrete target machines, and to completely verify them down to their binary machine code implementations.

This document describes the complete compiler construction in high level implementation language ComLisp for a four-phase compilation transforming ComLisp–programs to binary machine code executables on transputer T400 processors. The compilation is modularized to four steps using three intermediate languages, a stack language, a C-like abstract machine oriented language, and an assembly language. Compiling specifications between each pair of source and target languages are given as inductively defined relations. In this report we describe, how these specifications are refined to a system of first order mutually recursive ComLisp–functions, i.e. to a compiler program in high level implementation language ComLisp.

The correctly constructed ComLisp–compiler program, proved to be compliant to the compiling specification, may be used in order to bootstrap itself as a binary transputer–machine code executable. The bootstrapping process, however, initially depends on an unverified execution basis for ComLisp. Without further investigation it does not guarantee full correctness. We depend on unverified tool support if we use a CommonLisp system in order to execute this compiler. Mathematical a-posteriori control (double checking) of the bootstrapping result will close this gap for the fully verified initial ComLisp–compiler executable. However, the compiling specification and its correctness proof with respect to source and target language semantics (compiling verification) as well as the documentation of the a-posteriori double checking of the bootstrapping result are not given here, although the essential parts of the compiling specification are repeated in this document.

Keywords

compiler, compiler implementation, compiler implementation language, compiler construction, compiler verification, intermediate languages, implementation verification, implementation of Lisp, transputer–machine code

Zusammenfassung

Dieser Bericht ist ein Teil einer Reihe von Dokumenten, die zusammen die voll verifizierte Spezifikation, Konstruktion und Implementierung eines ComLisp-Übersetzers lückenlos dokumentieren. Die vorliegende Arbeit enthält die Beschreibung der Übersetzerkonstruktion in hoher Implementierungssprache ComLisp einer Vier-Lauf-Übersetzung von ComLisp über Stack-, C- und Assembler-Zwischensprachen in den ausführbaren binären Maschinencode des INMOS-transputer T400. ComLisp ist Teilsprache von ANSI-CommonLisp. Programme sind Systeme wechselseitig rekursiver Funktions- und Prozedurdefinitionen erster Ordnung. Die Übersetzungsspezifikation wird der Übersetzerimplementierung gegenübergestellt. Die Übersetzung ist durch Angabe induktiver Definitionen der zugrundeliegenden Übersetzungsrelationen spezifiziert. Die Verfeinerung in ein ComLisp-Programm ist Gegenstand dieses Berichts, so daß der ComLisp-Übersetzer in der Lage ist, sich selbst per Bootstrap in den transputer–Maschinencode zu übersetzen.

Die Arbeit ist im Rahmen des DFG-Forschungsprojektes *Verifix* über "Korrekte Übersetzer" entstanden, das das Ziel hat, Methoden zur Konstruktion korrekter Programmiersprachübersetzer für praktisch relevante Sprachen und reale Maschinen zu entwickeln und bis hinab zu ihrer binären Maschinencode-Implementierung lückenlos zu verifizieren. Mit dem hohen Korrektheitsanspruch in *Verifix* ist der oben beschriebene Bootstrap-Vorgang zunächst nicht lückenlos korrekt, da unverifizierte Hilfsmittel, z.B. ein CommonLisp-System, verwendet werden. Mathematische a-posteriori-Kontrolle des Bootstrap-Resultats schließt diese Lücke, so daß schließlich ein initiales, ablauffähiges voll verifiziertes ComLisp-Übersetzerprogramm entsteht.

Die Dokumentation der Übersetzerkonstruktion, die Spezifikation und Implementierung in hoher Implementierungssprache zueinander in Beziehung setzt, soll dem Compilerbauer einen detaillierten Überblick über die Compiler-Implementierung und die bei der Programmierung durchgeführten Verfeinerungsschritte geben. Die vorliegende Arbeit enthält die Dokumentation dieses Implementierungsschritts; Spezifikationen. Korrektheitsbeweise relativ zur Semantik der beteiligten Sprachen (Übersetzungsverifikation) und Dokumentation der a-posteriori-Überprüfung (Implementierungskorrektheitsbeweis durch Probe) sind nicht enthalten.

Schlüsselworte

Implementierungsverifikation, Lisp-Implementierung, System-Implementierungssprachen, Übersetzer, Übersetzerimplementierung, Übersetzerkonstruktion, Übersetzerverifikation, Übersetzungsspezifikation, transputer, Zwischensprachen

Contents

"You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code."

Ken Thompson
ACM Turing Award Lecture 1984

Chapter 1

Introduction

Motivation

Verifix is a German joint project on *Correct Compilers* [GDG+96]. Three research groups at the universities of Karlsruhe, Ulm and Kiel cooperate, funded by the DFG (Deutsche Forschungsgemeinschaft). The major goal is to develop methods for correct realistic compiler construction for practically relevant source languages and concrete target machines, and to completely verify them down to their binary machine code implementations. Thus, *Verifix* concentrates on the

- construction of correct compilers and their implementations,
- which generate efficient target code
- for realistic processors and languages,
- using practical verification methods and mechanical proof support.

The use of computer based systems in safety critical applications justifies and requires the verification of software components. Correct program execution, however, crucially depends on the correctness of the binary machine code executable, and, therefore, on the correctness of compiler programs. This is true for safety, but also for security. Ken Thompson [Tho84], the inventor of Unix, devoted his 1984 Turing Award Lecture to security problems due to Trojan horses intruded by compilers and compiler implementations.

L.M. Chirica and D.F. Martin [CM86] first explicitly distinguished verification of compiling specifications from verification of compiler implementations. J Moore [Moo88, Moo96] additionally stressed the need also for verification of the binary machine code executable of the compiler. In order to prove full compiler correctness, we have to verify both

- the compiling specification and
- the compiler implementation.

Verifix shows, that rigorous compiler verification down to the binary machine code executable of the compiler is possible and feasible [GDG+96, GGH+97, GH98a]. In order to trust compiler programs for system implementation, like compiler implementation itself, an initial fully correct

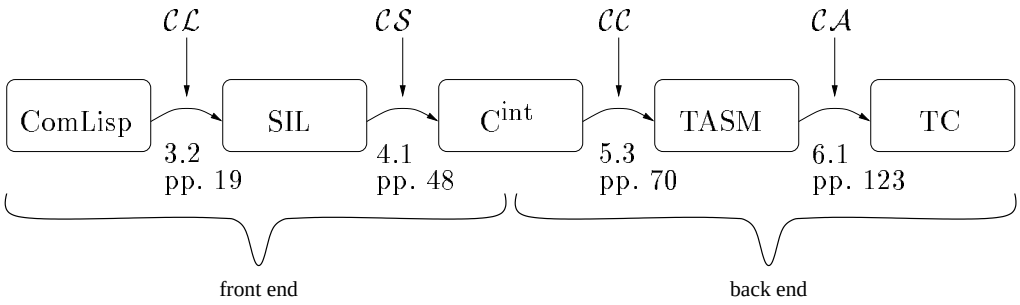
compiler executable has to be constructed, verified without depending on the correctness of any unverified tool. For this, we proceed in four steps, which correspond to four phases of a simple cascaded software engineering model [GH98a]:

- Define an appropriate notion of *correct compilation* for sequential imperative languages. It has to guarantee adequate correctness properties of source–target program pairs even for concrete target processors with their finite resource limitations (*preservation of partial correctness*).
- Define a compiling specification $\mathcal{C}$, a relation between source and target programs, and prove semantically, that it preserves partial correctness (*compiling verification*).
- Construct a compiler program $\pi_{\mathcal{C}}$ in the source language and prove, that $\pi_{\mathcal{C}}$ is a *refinement* (correct implementation) of $\mathcal{C}$ in the sense of preserving partial correctness (*correct compiler construction*).
- Use an existing (usually unverified) implementation of the source language to execute $\pi_{\mathcal{C}}$. Apply $\pi_{\mathcal{C}}$ to itself and bootstrap a compiler executable $m_{\mathcal{C}}$. Check syntactically, that $m_{\mathcal{C}}$ is correct, i.e. that it actually has been generated according to $\mathcal{C}$ (*compiler implementation verification*).

Preservation of partial program correctness [GDG+96, MO96, GMO96] is essential, adequate and sufficient for our purposes. Compiling verification proceeds according to M. Müller-Olm's approach in [MO97], compiler implementation verification according to U. Hoffmann's approach in [Hof98a]; a compiler implementation written in ComLisp is available, and the compiler bootstrap runs successfully on a concrete 1 MByte transputer T400 single board computer.

Overview

This report concentrates on correct compiler construction, the third item above. The major goal is to give a comprehensive and complete documentation of the entire compiler program, which is implemented in ComLisp and compiles ComLisp programs to the binary machine code of the INMOS transputer T400. We relate the compiler implementation module by module to the compiling specification which is defined in [GH98b]. The structure of the compiling specification and of the compiler program is shown in the following picture:



Consequently, this document is structured into four major parts as well. Section 3.2 in chapter 3 relates the ComLisp implementation of the compiler from ComLisp to SIL to the compiling

specification $\mathcal{CL}$ betweeen ComLisp and SIL. Section 4.1 in chapter 4 shows the compiler construction for the SIL-compiler w.r.t. $\mathcal{CS}$, section 5.3 in chapter 5 for C^{int} to TASM and $\mathcal{CC}$, and section 6.1 in chapter 6 for the assembler and $\mathcal{CA}$.

The program uses auxiliary ComLisp functions from the *runtime system* (chapter A), and the data and operation refinement for s-expressions (which takes place from SIL to C^{int}) produces a constant code part, the *core runtime system* (chaper B), a system of C^{int} procedures which implement the ComLisp (SIL) operators.

The compiler implementation in high level implementation language ComLisp is a refinement (correct implementation) of the compiling specification. The specification is an explicit inductive definition of a compiling relation, which maps source to target programs and possibly is non-deterministic and non-constructive. On the other hand, the compiler program defines a deterministic function mapping s-expression representation of source programs explicitely and constructively to s-expression representations of target programs. The correct construction of this deterministic function from the specification is called *correct compiler construction* above (third item), and it has to preserve partial correctness [GH98a]. This means, that we have to show, that whenever the deterministic compiler function or system of functions (the program) returns a result for a given s-expression representation of an argument, that this result is the representation of one of the specified target code fragments, and hence is a correct result. ComLisp, SIL, C^{int}, TASM, and TC all have s-expression syntaxes. Therefore, input and output data representation is easy, because the specification already maps s-expressions to s-expressions.

By far the largest part of this document is generated mechanically from the compiling specification [GH98b] and the program source of the compiler implementation in ComLisp. Although most comments, remarks and related topics are written manually, the specification types, and the specification extracts and code on those pages with a "Code Review" page style are automatically generated.

This document can be read as a full compiler documentation including the entire compiler code or as a code review document for the compiler implementation. But it also an informal mathematical proof, that the compiler implementation in high level implementation language ComLisp refines (correctly implements) the compiling specification in the sense of preservation of partial correctness. Taken as a proof document, however, there are two important points missing in this report:

The first is the correctness of the data and operation refinement step for the representation of Lisp-s-expressions in the linear C^{int} memory, which takes place in the compilation from SIL to C^{int}. The *core runtimesystem* (a system of C^{int} code modules, cf. chapter B) implements the SIL (or ComLisp) operators, and the proof that this operation refinement preserves partial correctness is not included here. The same is true for the correctness proof of the code which computes the initial heap segment representing the SIL program constants.

The second point is the correctness of the ComLisp reader and printer. They are both part of the *runtimesystem code* (cf. chapter A) which is written in ComLisp and is part of every compiler program. Correctness of of the read and print functions will be specified on the basis of classical definitions of the lexical structure and a context free attribute grammer. This specification and the program correctness proof for the implemementation will be part of other documents.

Acknowledgements

The work presented here is the result of intensive research and many experiments and discussions. Many details are influenced by compromises between mathematical compiling verification on the one hand and by requirements from (correct) compiler construction and implementation verification on the other hand. Many thanks to the local *Verifix* group here at Kiel, in particular to Dagobert Michelsen and to Jürgen Rühle for their contribution to compiling specification and compiler implementation. Dagobert Michelsen has contributed to the data and operation refinement from SIL to C^{int} and has written an earlier version of the core runtime system. Jürgen Rühle has written the final version of the boot loader program and much of an earlier version of the C^{int} to TASM–compiler. And we would like to thank Hans Langmaack, for making this research possible, for his critical guidance, and for lots of inspiring ideas.

Chapter 2

Main Programs and General Structure

This chapter includes the *main program functions* (those parameterless functions, which are called by the main program) of five compiler programs. Four separate standalone compiler programs are used to translate ComLisp to SIL, SIL to C^{int}, C^{int} to TASM, and TASM to TC. These compilers use four different main program functions and consist of the compiling functions which are defined separately in the following chapters. All of them share the runtimesystem and the core runtimesystem code. The construction of these compilers is just a copy and paste, they even could be identical except for the main program function. However, it is not necessary for one compiler to contain code of the others which is not called, anyway.

The bootstrapping procedure is described in detail in [Hof98a]. The idea is to use syntactical code checking and to exploit the following *diagonal argument*: First, use an untrusted execution basis for the four ComLisp compiler implementations in order to subsequently execute and apply them to the TASM to TC standalone compiler. Use syntactical code checking to assure that every intermediate compiler result and in particular the binary target code is in accordance to the verified specification. This produces a first fully correct executable for the TASM to TC compiler which now can be used to correctly assemble the further compiler phases. Apply the same procedure in order to produce a fully checked (now) TASM implementation of the C^{int} to TASM compiler. Use the verified assembler executable in order to translate this TASM program to TC. And so on. Finally we get four fully verified compiler executables written in machine code TC. These four programs may now be used to bootstrap the desired (fifth) complete compiler implementation.

The entire code is included in this fifth compiler, which directly translates from ComLisp to the binary transputer machine code TC. There are again two main program functions. One is the desired final compiler, which translates ComLisp programs to TC without any further printing of intermediate results. The other one may be used to bootstrap the compiler implementation by only one execution of the compiler applied to itself, in order to perform *strong compiler test* and *code inspection* as described e.g. in [Lan97a, Hof98a].

2.1 Standalone Programs

Name: CL to TC Standalone

Part of: Main Program

Specification:

These functions are not explicitly specified.

Source–Code:

```
(defun CL–standalone ()
  (print–sequence (CL (read–sequence)))
  (write–eof))

(defun CS–standalone ()
  (print–sequence (CS (read–sequence)))
  (write–eof))

(defun CC–standalone ()
  (print–sequence (CC (read–sequence)))
  (write–eof))

(defun CA–standalone ()
  (print–hexadecimal (CA (read–sequence)))
  (write–eof))
```

Issues:

Main program functions.

Comments:

These four functions are the main program functions of the four separate standalone compiler programs which translate ComLisp to SIL, SIL to C^{int}, C^{int} to TASM, and TASM to TC.

Name: CL to TC Standalone

Part of: Main Program

Specification:

This function is not explicitly specified.

Source–Code:

```
(defun CL2TC ()
  (print-hexadecimal
   (CA
    (print-sequence
     (CC
      (print-sequence
       (CS
        (print-sequence
         (CL
          (print-sequence (read-sequence)))))))))
  (write-eof))


(defun CL2TC ()
  (print-hexadecimal
   (CA (CC (CS (CL (read-sequence))))))
  (write-eof))
```

Issues:

Main program functions.

Comments:

These functions are two different main program functions for the entire compiler from ComLisp to the transputer machine code TC. They read the source program, apply the functional composition of the four compiler functions and print the target program.

The first main program function additionally prints the source and the resulting intermediate and target programs. This enables the *strong compiler test* as described e.g. in [Lan97a], which allows for saving the code inspection work for the Lisp-reader, because the printer is completely checked and thus is guaranteed to be correct and correctly implemented.

Chapter 3

ComLisp to SIL

The compiling relation between ComLisp and SIL is specified in [GH98b] to be a syntactical mapping between ComLisp source programs and SIL target programs. It is defined recursively using several auxiliary relations which correspond to the ComLisp source program structure. The types are as follows:

$$
\begin{array}{rcl}
\mathcal{CL} & \in & \texttt{<program>} \rightharpoonup \texttt{<program>}_{\text{SIL}} \\
\mathcal{CL}_{\text{decl}} & \in & \texttt{<declarations>} \rightharpoonup \mathit{env} \times \texttt{<declarations>}_{\text{SIL}} \times \texttt{<definition>}_{\text{SIL}}{}^{*} \\
\mathcal{CL}_{\text{def}} & \in & \texttt{<toplevelform>} \times \mathit{env} \rightharpoonup \texttt{<definition>}_{\text{SIL}} \\
\mathcal{CL}_{\text{form}} & \in & \texttt{<form>} \times \mathit{env} \times \mathit{env} \times \mathbb{N}_0 \rightharpoonup \texttt{<form>}_{\text{SIL}}{}^{*} \\
\mathcal{CL}_{\text{forms}} & \in & \texttt{<form>}^{*} \times \mathit{env} \times \mathit{env} \times \mathbb{N}_0 \rightharpoonup \texttt{<form>}_{\text{SIL}}{}^{*} \\
\mathcal{CL}_{\text{progn}} & \in & \texttt{<form>}^{*} \times \mathit{env} \times \mathit{env} \times \mathbb{N}_0 \rightharpoonup \texttt{<form>}_{\text{SIL}}{}^{*} \\
\mathcal{CL}_{\text{cond}} & \in & (\texttt{<form>}^{*})^{*} \times \mathit{env} \times \mathit{env} \times \mathbb{N}_0 \rightharpoonup \texttt{<form>}_{\text{SIL}}{}^{*}
\end{array}
$$

where

$$
\mathit{env} = \texttt{<ident>} \xrightarrow{\text{part}} \texttt{<integer>}
$$

These relations are now compared to ComLisp implementations, to a system of mutually recursive partial function definitions on Lisp-s-expression representations of the corresponding syntactical domains. The general procedure of this *correct compiler construction* is to define partial functions on the original domains which refine the relations, and then use a standard data refinement step in order to implement these partial functions by ComLisp functions. However, the data refinement is obvious (see below) and in most cases we can just compare the specified relations directly with the corresponding ComLisp functions so as to avoid the additional formalization of the partial functions.

In some cases, however, the specification defines a proper relation, either explicitly or because it refers to non-constructive side conditions which allow for different solutions. In those cases, where the ComLisp functions are non-trivial proper refinements of the specification, the construction steps will be described in more detail.

Our notation (cf. [Hof98a, GH98b]) above already suggests inputs and outputs of the corresponding partial functions. Additional recursive functions are necessary in order to implement the linear recursions on sequences of statements or expressions. These linear recursion is not formalized in the specifications. Instead, we used a ...-notation whenever adequate. Furthermore, in many cases standard program transformations like those defined in [Par90] have been applied in order to eliminate linear and tail recursion.

3.1 Lisp Data Representation

The specification uses Lisp-s-expression syntaxes for source an target program fragments. Thus, program fragments are s-expressions. The auxiliary domains used in the specification, however, like e.g. environments, are not yet s-expressions and have to be represented by Lisp-s-expressions. We use an obvious data (and operation) refinement, which represents sequences and n-tuples by the corresponding lists, and finite mappings by so called *assiciation lists*. Natural numbers, integers, and characters are left unchanged.

Cartesian Products $<d_1, \ldots, d_n> \in D_1 \times \ldots \times D_n$ are represented by n-element true lists `(`d'_1 ... d'_n`)` (ending with `NIL` as final `CDR`) of the corresponding elements. The functions `LIST` is used in order to construct n-tuples, selection is expressed by the corresponding `CAR`-`CDR`-chains.

Sequences $d_1 \ldots d_n \in D^*$ are represented by true lists `(`d'_1 ... d'_n`)` as well, again using `LIST` as constructor and `CAR` and `CDR` as selectors.

Finite Mappings like e.g. environments $\rho \in D_1 \xrightarrow{\text{part}} D_2$ are represented by association lists `((`$d'_{1,1}$. $d'_{2,1}$`)` ... `(`$d'_{1,n}$. $d'_{2,n}$`))`. The functions `ASSOC` or `ASSOC-EQUAL` are used in order to find the result $d'_{2,j}$ for an argument $d'_{1,j}$. This implements function application. The functions `CONS` and `LIST` or `LIST*` are used in order to construct or to modify these mappings.

`ASSOC` and `ASSOC-EQUAL` are total functions on true association lists (though they are partial on arbitrary data, of course). They return `NIL` if the original function has been undefined on a particular argument, the pair `(`$d'_{1,j}$. $d'_{2,j}$`)` otherwise. Therefore, at a first sight, implementing function application by `ASSOC` does not preserve partial correctness. However, this is not really true. It is only because we did not explicitly distinguish between *non-definedness* and well-defined *unbound* values in the specification. We used partial functions also in order to express total functions which only finitely often return a non *unbound* value. The latter would also be represented by finite association lists, of course, allowing `ASSOC` to return `NIL` for this *unbound* value, as it does. Whenever the specification refers to *non-definedness* of an environment on a particular argument, this actually means *unboundness*.

Note, that in many cases the compiling relation is specified to return one element target code sequences. This will not be clear from the specification part itself. We have to consider the corresponding types as well. However, the ComLisp implementation of course has to construct one elements lists in these cases.

3.2 ComLisp–Compiler: Code Review

Name: CL

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}[\![\, d\ t_1\ t_2\ \ldots\ t_n \,]\!] \supseteq \begin{array}{l} d' \\ main' \\ \mathcal{CL}_{\mathrm{def}}[\![\, \delta_1 \,]\!]\,\gamma \\ \mathcal{CL}_{\mathrm{def}}[\![\, \delta_2 \,]\!]\,\gamma \\ \ldots \\ \mathcal{CL}_{\mathrm{def}}[\![\, \delta_m \,]\!]\,\gamma \end{array}$$

---

where $d =$ (*funs vars main syms qc*) $\in$ <`declarations`> and
$t_1,\ t_2,\ \ldots,\ t_n \in$ <`toplevelform`> and $d\ t_1\ t_2\ \ldots\ t_n$ is wellformed.

The resulting SIL–declarations d', the SIL–main–procedure $main'$ and the global environment γ, which maps global variable identifiers to absolute addresses, are defined by $\mathcal{CL}_{\mathrm{decl}}[\![\, d \,]\!] \supseteq <\gamma, d', main'>$

$\delta_1,\ \delta_2,\ \ldots,\ \delta_m$ are the program function definitions, i. e. the top-level forms of shape `(DEFUN ...)`.

$\mathcal{CL}.1$

Source–Code:

```
(defun CL (p)
  (let* ((decls (car p))
         (defs (cdr p))
         (env–decl–defs (CLdecl decls))
         (env (car env–decl–defs))
         (decl–defs (cdr env–decl–defs)))
    (append decl–defs (CLdefs defs env))))
```

Comments:

The function `CLdefs` is applied to $t_1\ t_2\ \ldots\ t_n$ instead of $\delta_1\ \delta_2\ \ldots\ \delta_m$, but it returns only the compiled function definitions. Non defining top level forms are collected by `CLdecl`.

Related Documents: Function `CLdefs`.

Name: CLdecl

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{decl}}[\![\,(\mathit{funs}\ \mathit{vars}\ \mathit{main}\ \mathit{syms}\ \mathit{qc})\,]\!] \supseteq \langle\gamma,\ (|\mathit{vars}|\ \mathit{funs'}\ \mathit{syms}\ \mathit{qc}),\ \mathcal{CL}_{\text{def}}[\![\,\texttt{(DEFUN \_main () }\mathit{main'}\texttt{)}\,]\!]\gamma\rangle$$

---

where these declarations are wellformed, *funs*, *vars* $\in$ <`ident`>*, *syms* $\in$ <`symbol`>*, *main'* $\in$ <`form`>*, and *qc* $\in$ <`s-expr`>*. *main'* $= \mathit{tlf}_1 \ldots \mathit{tlf}_\nu$, $\nu \geq 0$, $\mathit{tlf}_i \in$ <`top-level-form`> is the sequence of non-defining top level forms in the list *main* = (*tlf*$_1$ ... *tlf*$_\nu$).
γ must be an injection from *vars* to the positive integer numbers >= 1, *funs'* is *funs* plus the identifier `_main`.

$\mathcal{CL}.2$

Source–Code:

```
(defun CLdecl (decls)
  (let* ((funs (cons '_MAIN (car decls)))
         (globals (cadr decls))
         (tlfs (caddr decls))
         (syms (cadddr decls))
         (fats (cadr (cdddr decls)))
         (main  (CLdef (list* 'DEFUN '_MAIN () tlfs) globals)))
    (list globals
          (list (list-length globals) funs syms fats)
          main)))
```

Comments:

The list *qc* of fat quote constants is called `fats` in the program, *funs'* is called `funs`.

Name: CLdefs

Part of: ComLisp to SIL–compiler

Specification:

Source–Code:

```
(defun CLdefs (defs genv)
  (let* ((result nil))
    (do ()
        ((null defs))
      (if (eql (caar defs) 'DEFUN)
          (setq result (cons (CLdef (car defs) genv) result)))
      (setq defs (cdr defs)))
    (reverse result)))
```

Issues:

This function implements the implicit linear recursion, that appears in the specification of the transformation of sequences of ComLisp definitions

$$\delta_1 \ \ldots \ \delta_n \quad \text{into sequences} \quad \mathcal{CL}_{\text{def}}[\![\,\delta_1\,]\!]\,\gamma \ \ldots \ \mathcal{CL}_{\text{def}}[\![\,\delta_n\,]\!]\,\gamma\,.$$

of compiled definitions. Sequences are implemented as lists, and `genv` holds the representation of the global environment γ.

Comments:

The linear recursion is transformed to a tail recursion which is eliminated using standard program transformations [Par90]. The resulting code is a loop. Note, that `CLdefs` is not applied to the list of function definitions, but to the list of top level forms. Therefore, it compiles and returns only those forms from `defs`, which are function definitions, i.e. start with `DEFUN`.

The loop constructs the list of compiled definitions in reverse order. Therefore, the final result is `(reverse result)`.

Name: CLdef

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{def}}[\![\texttt{(DEFUN } p \texttt{ (}p_1 \ldots p_n\texttt{) } f_1 \ldots f_m\texttt{)}]\!]\,\gamma \;\supseteq\; \texttt{(DEFUN } p \\ \mathcal{CL}_{\text{progn}}[\![f_1 \;\ldots\; f_m]\!]\,\rho\;\gamma\;n \\ \texttt{(\_COPY } n\ 0\texttt{))}$$

---

where $p,\ p_1\ \ldots,\ p_n \in$ <symbol> and $f_1,\ \ldots,\ f_m\ \in$ <form>.

In SIL relative addresses are represented by non–negative numbers ($\mathbb{N}_0$). The local variable environment ρ must be a bijection from $\{p_1,\ \ldots,\ p_n\}$ to a beginning interval of the non negative numbers $[0, n-1]$.

$\mathcal{CL}.3$

Source–Code:

```
(defun CLdef (d genv)
  (let* ((p          (cadr d))
         (ps         (caddr d))
         (fs         (cdddr d))
         (n          (list–length ps)))
    (list* 'DEFUN p
           (append (CLprogn fs (get–indexed ps 0) genv n)
                   (list (list '_COPY n 0))))))
```

Comments:

The function `gen-indexed` is used in order to construct the local variable environment ρ from the list of formal parameters `(`$p_1 \ldots p_n$`)`. Variable indices are the consecutive integers between 0 and $n-1$.

Related Documents: Function `get-indexed`.

Name: get-indexed

Part of: ComLisp to SIL–compiler

Specification:

Source–Code:

```
(defun get–indexed (varlist index)
  (if (null varlist)
      NIL
      (cons (cons (car varlist) index)
            (get–indexed (cdr varlist) (+ 1 index)))))
```

Issues:

Given a variable list $(p_1 \ldots p_n)$ and a start index $i \in \mathbb{N}_0$, this function constructs a variable environment $((p_1 \ . \ i) \ \ldots \ (p_n \ . \ i+n-1))$.

Name: CLform

Part of: ComLisp to SIL–compiler

Specification:

This function is specified by a collection of rules each of which separately defines the compilation of a particular ComLisp form (statement and/or expression). In the following we will compare each of the rules separately with the code occuring in the single cases of the conditional refining the relation $\mathcal{CL}$ below. The construction of this function is described in detail in [Hof98a].

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list-length args)))
       (cond
         ;; IF ------------------------------------------------------------
         ((eql key 'IF) ... )
         ;; OR ------------------------------------------------------------
         ((eql key 'OR) ... )
         ;; AND-----------------------------------------------------------
         ((eql key 'AND) ... )
         ;; COND----------------------------------------------------------
         ((eql key 'COND) ... )
         ;; PROGN---------------------------------------------------------
         ((eql key 'PROGN) ... )
         ;; LET-----------------------------------------------------------
         ((eql key 'LET) ... )
         ;; LET*----------------------------------------------------------
         ((eql key 'LET*) ... )
         ;; DO------------------------------------------------------------
         ((eql key 'DO) ... )
         ;; SETQ----------------------------------------------------------
         ((eql key 'SETQ) ... )
         ;; LIST----------------------------------------------------------
         ((eql key 'LIST) ... )
         ;; LIST*---------------------------------------------------------
         ((eql key 'LIST*) ... )
         ;; QUOTE---------------------------------------------------------
         ((eql key 'QUOTE) ... )
         ;; Application---------------------------------------------------
         (T ... ))))

    ;; Literal-------------------------------------------------------------
    ((integerp form)   ... )
    ((characterp form) ... )
    ((null form)       ... )
    ((stringp form)    ... )
    ((eql form T)      ... )
```

```
;; variable identifier————————————————————————————————————————
((assoc form lenv)  ... )
((member form genv) ... )
(T
 (errorstop 1))))
```

Name: CLform IF–translation

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{form}}[\![\texttt{(IF } f_1\ f_2\ f_3\texttt{)}]\!]\,\rho\,\gamma\,k \supseteq \mathcal{CL}_{\text{form}}[\![f_1]\!]\,\rho\,\gamma\,k\ \texttt{(IF } k\ \texttt{(}\mathcal{CL}_{\text{form}}[\![f_2]\!]\,\rho\,\gamma\,k\ \texttt{)}\ \texttt{(}\mathcal{CL}_{\text{form}}[\![f_3]\!]\,\rho\,\gamma\,k\ \texttt{))}$$

$$\mathcal{CL}_{\text{form}}[\![\texttt{(IF } f_1\ f_2\texttt{)}]\!]\,\rho\,\gamma\,k \supseteq \mathcal{CL}_{\text{form}}[\![f_1]\!]\,\rho\,\gamma\,k\ \texttt{(IF } k\ \texttt{(}\mathcal{CL}_{\text{form}}[\![f_2]\!]\,\rho\,\gamma\,k\ \texttt{)}\ \texttt{((\_COPYC NIL } k\texttt{)))}$$

where $f_1,\ f_2,\ f_3 \in$ <form>.

$\mathcal{CL}.4$

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list-length args)))
       (cond
         ;; IF ------------------------------------------------------------
         ((eql key 'IF) ... )
         (if (null (cddr args))
             ;; one way  (IF f1 f2) -> f1' (IF k (f2') ((_COPYC NIL k)))
             (let ((f1 (car args))
                   (f2 (cadr args)))
               (append
                (CLform f1 lenv genv k)
                (list
                 (list 'IF k
                       (CLform f2 lenv genv k)
                       (list (list '_COPYC NIL k))))))
             ;; two way  (IF f1 f2 f3) -> f1' (IF k (f2') (f3'))
             (let ((f1 (car args))
                   (f2 (cadr args))
                   (f3 (caddr args)))
               (append
                (CLform f1 lenv genv k)
                (list
                 (list 'IF k
                       (CLform f2 lenv genv k)
```

```
                (CLform f3 lenv genv k))))))
    ...
    )))))
```

Name: CLform AND/OR–translation

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{form}}[\![\texttt{(OR } f_1\ f_2\texttt{)}]\!]\,\rho\ \gamma\ k \ \supseteq\ \mathcal{CL}_{\text{form}}[\![f_1]\!]\,\rho\ \gamma\ k$$
$$\texttt{(IF } k\ \texttt{() (}\ \mathcal{CL}_{\text{form}}[\![f_2]\!]\,\rho\ \gamma\ k\ \texttt{))}$$

$$\mathcal{CL}_{\text{form}}[\![\texttt{(AND } f_1\ f_2\texttt{)}]\!]\,\rho\ \gamma\ k \ \supseteq\ \mathcal{CL}_{\text{form}}[\![f_1]\!]\,\rho\ \gamma\ k$$
$$\texttt{(IF } k\ \texttt{(}\ \mathcal{CL}_{\text{form}}[\![f_2]\!]\,\rho\ \gamma\ k\ \texttt{) ())}$$

where $f_1,\ f_2 \in$ <form>.

$\mathcal{CL}.5$

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list-length args)))
       (cond
        ...
        ;; OR ------------------------------------------------------------
        ((eql key 'OR)
         ;; (OR f1 f2) -> f1' (IF k () (f2'))
         (let ((f1 (car args))
               (f2 (cadr args)))
           (append
            (CLform f1 lenv genv k)
            (list (list 'IF k NIL (CLform f2 lenv genv k))))))
        ;; AND-----------------------------------------------------------
        ((eql key 'AND)
         ;; (AND f1 f2) -> f1' (IF k (f2') ())
         (let ((f1 (car args))
               (f2 (cadr args)))
           (append
            (CLform f1 lenv genv k)
            (list (list 'IF k (CLform f2 lenv genv k) NIL))))
        ...
        )))
    ...
    ))
```

Related Documents: Definition of CLform on page 25

Name: CLforms

Part of: ComLisp to SIL–compiler

Source–Code:

```
(defun CLforms (fs lenv genv k)
  (let ((y NIL))
    (do ()
        ((null fs))
      (setq y (append y (CLform (car fs) lenv genv k)))
      (setq fs (cdr fs))
      (setq k (+ k 1)))
    y))
```

Issues:

This function implements the implicit linear recursion, that appears in the specification of sequences of ComLisp forms

$$f_1 \ \ldots \ f_n \quad \text{into sequences} \quad \mathcal{CL}_{\text{form}}[\![f_1]\!] \rho \ \gamma \ k \ \ldots \mathcal{CL}_{\text{form}}[\![f_n]\!] \rho \ \gamma \ k+n-1 \, .$$

of compiled forms. Sequences are implemented as lists, and the resulting target code lists have to be appended. `lenv`, `genv`, and `k` hold the local and global environment and stack position, representation of ρ, γ, and k, respectively.

Comments:

The linear recursion is transformed to a tail recursion which is eliminated using standard program transformations [Par90]. The resulting code is a loop.

Related Documents: Definition of `CLform` on page 25

Name: CLform translation of function applications

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{form}}[\![\,(p\ f_1\ \ldots\ f_n)\,]\!]\,\rho\ \gamma\ k\ \supseteq\ \begin{array}{l}\mathcal{CL}_{\text{form}}[\![\,f_1\,]\!]\,\rho\ \gamma\ k\\ \ldots\\ \mathcal{CL}_{\text{form}}[\![\,f_n\,]\!]\,\rho\ \gamma\ k{+}n{-}1\\ (p\ k)\end{array}$$

where $p \in$ <ident>, and $f_1, \ldots, f_n \in$ <form>.

$\mathcal{CL}.6$

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list-length args)))
       (cond
        ...
        (T
         ;; (f f1 ... fn) -> f1' ... fn' (f k)
         (append
          (CLforms args lenv genv k)
          (list (list key k)))))))
    ...
    ))
```

Related Documents: Definition of `CLform` on page 25

Name: CLform LET–translation

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{form}}[\![\ \texttt{(LET (}\ (v_1\ \ i_1)\ (v_2\ \ i_2)\ \vdots\ (v_m\ \ i_m))\ f_1 \ldots f_n)\ \ ]\!]\rho\gamma k \quad \supseteq$$

$$\mathcal{CL}_{\text{form}}[\![\ i_1\ ]\!]\rho\gamma k$$
$$\mathcal{CL}_{\text{form}}[\![\ i_2\ ]\!]\rho\gamma\ \ k+1$$
$$\vdots$$
$$\mathcal{CL}_{\text{form}}[\![\ i_m\ ]\!]\rho\ \ \gamma\ \ k+m-1$$
$$\mathcal{CL}_{\text{progn}}[\![\ f_1 \ldots f_n\ ]\!]\rho[v_1 \mapsto k] \ldots [v_m \mapsto k+m-1]\ \ \gamma\ \ k+m$$
$$\texttt{(\_COPY}\ \ k+m\ \ k\texttt{)}$$

---

where $v_1, \ldots, v_m \in$ <ident> and $i_1, \ldots, i_m,\ f_1, \ldots, f_n \in$ <form>, $m, n \geq 0$.

$\mathcal{CL}.7$

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list–length args)))
       (cond
        ...

        ;; LET————————————————————————————————————————————————————
        ((eql key 'LET)
         ;; (LET ((v1 i1)...) f1...)
         (append
          (CLlet (car args) (cdr args) lenv genv lenv k)
          (list (list '_COPY (+ k (list–length (car args))) k))))
        ...
        )))
    ...
    ))
```

Source–Code:

```
(defun CLlet (bindings body lenv genv benv k)
  (if (null bindings)
      (CLprogn body benv genv k)
```

```
(let* ((v1 (caar bindings))
       (i1 (cadar bindings))
       (vs (cdr bindings)))
  (append
   (CLform i1 lenv genv k)
   (CLlet vs body lenv genv (cons (cons v1 k) benv) (+ k 1))))))
```

Comments:

The function `CLlet` is called; the final `COPY`-command is then added to the resulting code. `CLlet` implements all three linear recursions denoted by our ...-notation above at once. A *binding environment* `benv` accumulates the local bindings, whereas `CLlet` compiles every form i_j of the `LET` body within the original local environment `lenv`. This make the difference to `CLlet*`, which compiles these forms in the so far accumulated new binding environment `benv`.

Related Documents: Definition of `CLform` on page 25

Name: CLform LET*–translation

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{form}}[\![\ \texttt{(LET* (}\ (v_1\ i_1)\ (v_2\ i_2)\ \vdots\ (v_m\ i_m))\ f_1 \ldots f_n)\ ]\!]\rho\gamma k \quad \supseteq$$

$$\begin{array}{l}
\mathcal{CL}_{\text{form}}[\![\ i_1\ ]\!]\rho\gamma k \\
\mathcal{CL}_{\text{form}}[\![\ i_2\ ]\!]\rho[v_1 \mapsto k]\ \ \gamma\ \ k{+}1 \\
\quad\ \vdots \\
\mathcal{CL}_{\text{form}}[\![\ i_m\ ]\!]\rho[v_1 \mapsto k] \ldots [v_{m-1} \mapsto k{+}m{-}2]\ \ \gamma\ \ k{+}m{-}1 \\
\mathcal{CL}_{\text{progn}}[\![\ f_1 \ldots f_n\ ]\!]\rho[v_1 \mapsto k] \ldots [v_m \mapsto k{+}m{-}1]\ \ \gamma\ \ k{+}m \\
\texttt{(_COPY}\ \ k{+}m\ \ k\texttt{)}
\end{array}$$

---

where $v_1, \ldots, v_m \in$ <ident> and $i_1, \ldots, i_m,\ f_1, \ldots, f_n \in$ <form>, $m, n \geq 0$.

$\mathcal{CL}.8$

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list-length args)))
       (cond
         ...
         ;; LET*-------------------------------------------------------------
         ((eql key 'LET*)
          ;; (LET* ((v1 i1)...) f1...)
          (append
           (CLlet* (car args) (cdr args) lenv genv lenv k)
           (list (list '_COPY (+ k (list-length (car args))) k))))
         ...
         )))
    ...
    ))
```

Source–Code:

```
(defun CLlet* (bindings body lenv genv benv k)
  (if (null bindings)
      (CLprogn body benv genv k)
      (let* ((v1 (caar bindings))
```

```
        (i1 (cadar bindings))
        (vs (cdr bindings)))
    (append
     (CLform i1 benv genv k)
     (CLlet* vs body lenv genv (cons (cons v1 k) benv) (+ k 1))))))
```

Comments:

The function `CLlet*` is called; the final `COPY`-command is then added to the resulting code. `CLlet*` implements all three linear recursions denoted by our ...-notation above at once. A *binding environment* `benv` accumulates the local bindings. `CLlet` compiles every form i_j of the `LET` body within the new binding environment `benv`. This make the difference to `CLlet`, which compiles these forms in the original local environment `lenv`.

Related Documents: Definition of `CLform` on page 25

Name: CLform LIST/LIST*–translation

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{form}}[\![\texttt{(LIST } f_1 \ \ldots \ f_n\texttt{)}]\!]\,\rho\ \gamma\ k \ \supseteq\ \mathcal{CL}_{\text{form}}[\![f_1]\!]\,\rho\ \gamma\ k$$
$$\ldots$$
$$\mathcal{CL}_{\text{form}}[\![f_n]\!]\,\rho\ \gamma\ k{+}n{-}1$$
$$\texttt{(\_COPYC NIL } k{+}n\texttt{) (LIST* } n{+}1\ k\texttt{)}$$

$$\mathcal{CL}_{\text{form}}[\![\texttt{(LIST* } f_1 \ \ldots \ f_n\texttt{)}]\!]\,\rho\ \gamma\ k \ \supseteq\ \mathcal{CL}_{\text{form}}[\![f_1]\!]\,\rho\ \gamma\ k$$
$$\ldots$$
$$\mathcal{CL}_{\text{form}}[\![f_n]\!]\,\rho\ \gamma\ k{+}n{-}1$$
$$\texttt{(LIST* } n\ k\texttt{)}$$

where $f_1, \ldots, f_n \in$ <form>.

$\mathcal{CL}.9$

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list-length args)))
       (cond
         ...
         ;; LIST----------------------------------------------------------------
         ((eql key 'LIST)
          ;; (LIST f1 ... fn) -> f1' ... fn' (_COPYC NIL k+n) (LIST* n+1 k)
          (append
           (CLforms args lenv genv k)
           (list
            (list '_COPYC 'NIL (+ k n))
            (list 'LIST* (+ n 1) k))))
        ;; LIST*---------------------------------------------------------------
        ((eql key 'LIST*)
         ;; (LIST* f1 ... fn) -> f1' ... fn' (LIST* n k)
         (append
          (CLforms args lenv genv k)
          (list (list 'LIST* n k))))
         ...
         )))
    ... ))
```

Comments:

The difference between `LIST` and `LIST*` is that `LIST` appends a `NIL`, whereas `LIST*` uses f_n as final `CDR`. Thus, `(LIST` f_1 ... f_n`)` $\equiv$ `(LIST*` f_1 ... f_n `NIL)`.

Related Documents: Definition of `CLform` on page 25

Name: CLform SETQ–translation

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{form}}[\![\,\texttt{(SETQ}\ v\ f\texttt{)}\,]\!]\,\rho\ \gamma\ k \ \supseteq\ \mathcal{CL}_{\text{form}}[\![\,f\,]\!]\,\rho\ \gamma\ k \quad \text{where } \rho(v) \text{ defined}$$
$$\texttt{(\_COPY}\ k\ \rho(v)\ \texttt{)}$$

$$\mathcal{CL}_{\text{form}}[\![\,\texttt{(SETQ}\ v\ f\texttt{)}\,]\!]\,\rho\ \gamma\ k \ \supseteq\ \mathcal{CL}_{\text{form}}[\![\,f\,]\!]\,\rho\ \gamma\ k \quad \text{where } \rho(v) \text{ undefined and}$$
$$\texttt{(\_COPYG}\ k\ \gamma(v)\ \texttt{)} \qquad\qquad \gamma(v) \text{ defined}$$

---

where $v \in$ <symbol> and $f \in$ <form>.

$\mathcal{CL}$.10

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list-length args)))
       (cond
         ...
         ;; SETQ-------------------------------------------------------------
         ((eql key 'SETQ)
          ;; (SETQ v f) -> (_COPY k i) or (_COPYG k i)
          (let* ((v (car args))
                 (f (cadr args)))
            (cond
              ((assoc v lenv)
               (let ((i (cdr (assoc v lenv))))
                 (append
                  (CLform f lenv genv k)
                  (list (list '_COPY k i)))))
              ((member v genv)
               (let ((i (list-length (member v genv))))
                 (append
                  (CLform f lenv genv k)
                  (list (list '_COPYG k i)))))
              (T (errorstop 1)))))
         ...
         )))
    ...
    ))
```

Related Documents: Definition of CLform on page 25

Name: CLform variable–translation

Part of: ComLisp to SIL–compiler

Specification:

$\mathcal{CL}_{\text{form}}[\![v]\!] \rho\ \gamma\ k \supseteq$ `(_COPY` $\rho(v)$ k`)` where $\rho(v)$ defined

$\mathcal{CL}_{\text{form}}[\![v]\!] \rho\ \gamma\ k \supseteq$ `(_GCOPY` $\gamma(v)$ k`)` where $\rho(v)$ undefined and $\gamma(v)$ defined

---

where $v \in$ <ident>.

$\mathcal{CL}$.11

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ...
    ((assoc form lenv)
     (let ((i (cdr (assoc form lenv))))
       (list (list '_COPY i k))))
    ((member form genv)
     (let ((i (list–length (member form genv))))
       (list (list '_GCOPY i k))))
    ...
    ))
```

Related Documents: Definition of `CLform` on page 25

Name: CLprogn

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{form}}[\![\texttt{(PROGN } f_1 \ \ldots \ f_n\texttt{)}]\!]\,\rho\ \gamma\ k \ \supseteq\ \mathcal{CL}_{\text{progn}}[\![f_1 \ \ldots \ f_n]\!]\,\rho\ \gamma\ k$$

$$\mathcal{CL}_{\text{progn}}[\![f_1 \ \ldots \ f_n]\!]\,\rho\ \gamma\ k \ \supseteq\ \mathcal{CL}_{\text{form}}[\![f_1]\!]\,\rho\ \gamma\ k \quad \text{where } n \geq 1$$
$$\mathcal{CL}_{\text{form}}[\![f_2]\!]\,\rho\ \gamma\ k$$
$$\ldots$$
$$\mathcal{CL}_{\text{form}}[\![f_n]\!]\,\rho\ \gamma\ k$$

$$\mathcal{CL}_{\text{progn}}[\![\ ]\!]\,\rho\ \gamma\ k \ \supseteq\ \texttt{(\_COPYC NIL } k\texttt{)}$$

where $f_1, \ldots, f_n \in$ <form>.

$\mathcal{CL}$.12

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list-length args)))
       (cond
         ...
         ;; PROGN----------------------------------------------------------
         ((eql key 'PROGN)
          ;; (PROGN f1 ... fn)
          (CLprogn args lenv genv k))
         ...
         )))
    ...
    ))
```

Source–Code:

```
(defun CLprogn (forms lenv genv k)
  (cond
    ((null forms)
     (list (list '_COPYC NIL k)))
    ((eql (list-length forms) 1)
     (CLform (car forms) lenv  genv k))
    (T
     (append
      (CLform (car forms) lenv genv k)
```

```
(CLprogn (cdr forms) lenv genv k)))))
```

Issues:

The function `CLprogn` implements the linear recursion on the form list. `CLprogn` returns a list of SIL-forms. The first case in the conditional is the special case for the empty `PROGN`. Note: `CLform` could use `CLseq` for non empty `PROGN`s.

Related Documents: Definition of `CLform` on page 25

Name: CLcond

Part of: ComLisp to SIL–compiler

Specification:

$\mathcal{CL}_{\text{form}}[\![\ \texttt{(COND}\ c\texttt{)}\ ]\!]\rho\gamma k \supseteq \mathcal{CL}_{\text{cond}}[\![\ c\ ]\!]\rho\gamma k$ where $c \in \texttt{(<form>}^{+}\texttt{)}^{*}$

$\mathcal{CL}_{\text{cond}}[\![\ ]\!]\rho\gamma k \supseteq$ `(_COPYC NIL` k`)`

$\mathcal{CL}_{\text{cond}}[\![\ \texttt{(}p_1\ \ f_{11}\ldots f_{1m_1}\texttt{)}\ \ldots\ \texttt{(}p_n\ \ f_{n1}\ldots f_{nm_n}\texttt{)}\ ]\!]\rho\gamma k \supseteq$

$\mathcal{CL}_{\text{form}}[\![\ p_1\ ]\!]\rho\gamma k$
`(IF` k
$\quad$ `(`$\mathcal{CL}_{\text{form}}[\![\ f_{11}\ ]\!]\rho\gamma k\ \ldots\ \mathcal{CL}_{\text{form}}[\![\ f_{1m_1}\ ]\!]\rho\gamma k$`)`
$\quad$ `(`
$\quad\quad \vdots$
$\quad\quad \mathcal{CL}_{\text{form}}[\![\ p_n\ ]\!]\rho\gamma k$
$\quad\quad$ `(IF` k
$\quad\quad\quad$ `(`$\mathcal{CL}_{\text{form}}[\![\ f_{n1}\ ]\!]\rho\gamma k\ \ldots\ \mathcal{CL}_{\text{form}}[\![\ f_{nm_n}\ ]\!]\rho\gamma k$`)`
$\quad\quad$ `())... ))`

and the final (innermost) `IF`-expression may be empty, if $m_n = 0$.

---

where $n \geq 1, p_i, f_{ij} \in$ `<form>`.

$\mathcal{CL}$.13

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list-length args)))
       (cond
         ...
         ;; COND-------------------------------------------------------------
         ((eql key 'COND)
          ;; (COND cl1 ... cln)
          (CLcond args lenv genv k))
         ...
         )))
    ...
    ))
```

Source–Code:

```
(defun CLcond (cs lenv genv k)
  (cond
    ((null cs)
     (list (list '_COPYC NIL k)))
    ((null (cdr cs))
     (let ((p (caar cs))
           (fs (cdar cs)))
       (if (null fs)
           (CLform p lenv genv k)
           (append
            (CLform p lenv genv k)
            (list
             (list 'IF k (CLseq fs lenv genv k) NIL))))))
    (T
     (let ((p (caar cs))
           (fs (cdar cs)))
       (append
        (CLform p lenv genv k)
        (list
         (list 'IF k
               (CLseq fs lenv genv k)
               (CLcond (cdr cs) lenv genv k))))))))
```

Comments:

The construction of `CLcond` is described in detail in [Hof98a].

Related Documents: Definition of `CLform` on page 25

Name: CLseq

Part of: ComLisp to SIL–compiler

Source–Code:

```
(defun CLseq (forms lenv genv k)
  (cond
    ((null forms)
     NIL)
    (T
     (append
      (CLform (car forms) lenv genv k)
      (CLseq (cdr forms) lenv genv k)))))
```

Comments:

This function implements the linear recursion on lists of ComLisp-forms. It is similar to `CLprogn`, but is does not generate a `NIL` result in case the form list is empty.

Name: CLform DO–translation

Part of: ComLisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{form}}[\![\,\texttt{(DO () (}f\texttt{)}\ f_1\ \ldots\ f_n\texttt{)}\,]\!]\,\rho\,\gamma\,k \quad \supseteq \quad \texttt{(DO}\ k\ \texttt{(}\ \mathcal{CL}_{\text{form}}[\![\,f\,]\!]\,\rho\,\gamma\,k\ \texttt{)}\ \mathcal{CL}_{\text{progn}}[\![\,f_1\ \ldots\ f_n\,]\!]\,\rho\,\gamma\,k\ \texttt{)}\ \texttt{(\_COPYC NIL}\ k\texttt{)}$$

where $f_1, \ldots, f_n \in$ <form>.

$\mathcal{CL}$.14

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ((consp form)
     (let* ((key (car form))
            (args (cdr form))
            (n (list-length args)))
       (cond
        ...
        ;; DO----------------------------------------------------------------
        ((eql key 'DO)
         ;; (DO () (p) f1...) -> (DO (p') f1'...)
         (let ((p (caadr args))
               (fs (cddr args)))
           (list
            (list* 'DO k
                   (CLform p lenv genv k)
                   (CLprogn fs lenv genv k))
            (list '_COPYC NIL k))))
        ...
        )))
    ...
    ))
```

Related Documents: Definition of CLform on page 25

Name: CLform Constants

Part of: Comlisp to SIL–compiler

Specification:

$$\mathcal{CL}_{\text{form}}[\![\,\texttt{(QUOTE}\ s\texttt{)}\,]\!]\,\rho\ \gamma\ k \quad \supseteq \quad \texttt{(\_COPYC}\ s\ k\texttt{)}$$

where $s \in$ <s-expr>.

$\mathcal{CL}$.15

$$\mathcal{CL}_{\text{form}}[\![\,c\,]\!]\,\rho\ \gamma\ k \quad \supseteq \quad \texttt{(\_COPYC}\ c\ k\texttt{)}$$

where $c \in$ <integer> $\cup$ <character> $\cup$ <string> $\cup$ {NIL, T}.

$\mathcal{CL}$.16

Source–Code:

```
(defun CLform (form lenv genv k)
  (cond
    ...

      ;; (QUOTE f1) -> (_COPYC f1 k)
      ((equal key 'QUOTE)
       (list (list '_COPYC (car args) k)))

    ...

    ;; Literal------------------------------------------------------------
    ((integerp form)   (list (list '_COPYC form k)))
    ((characterp form) (list (list '_COPYC form k)))
    ((null form)       (list (list '_COPYC form k)))
    ((stringp form)    (list (list '_COPYC form k)))
    ((eql form T)      (list (list '_COPYC form k)))

    ...
    ))
```

Related Documents: Definition of CLform on page 25

Chapter 4

SIL to C^{int}

In this chapter we will compare the compiling relation between SIL and C^{int} as defined in [GH98b] with its implementation as a set of ComLisp functions. The compiling relation $\mathcal{CS}$ and its auxiliary relations have the following types:

$$
\begin{array}{rcl}
\mathcal{CS} & \in & \texttt{<program>}_{\text{SIL}} \rightharpoonup \texttt{<program>}_{C^{int}} \\
\mathcal{CS}_{\text{decl}} & \in & \texttt{<declarations>}_{\text{SIL}} \rightharpoonup \\
 & & \quad \textit{heapenv} \times \texttt{<declarations>}_{C^{int}} \\
 & & \quad \times \texttt{<data-definition>} \times \texttt{<definition>}^{*}{}_{C^{int}} \\
\mathcal{CS}_{\text{def}} & \in & \texttt{<definition>}_{\text{SIL}} \times \textit{heapenv} \rightharpoonup \texttt{<definition>}_{C^{int}} \\
\mathcal{CS}_{\text{form}} & \in & \texttt{<form>}_{\text{SIL}} \times \textit{heapenv} \rightharpoonup \texttt{<statement>}^{*} \\
\mathcal{CS}_{\text{heap}} & \in & \texttt{<program>}_{\text{SIL}} \times \textit{heapenv} \rightharpoonup \texttt{<integer>}^{*} \\
\mathcal{CS}_{\text{stack}} & \in & \texttt{<program>}_{\text{SIL}} \times \textit{heapenv} \rightharpoonup \texttt{<integer>}^{*}
\end{array}
$$

where

$$
\textit{heapenv} = \texttt{<sexpr>} \xrightarrow{\text{part}} (\texttt{<integer>} \times \texttt{<integer>})
$$

General remarks on our procedure and in particular on data representation have already been made in chapter 3. We will not repeat them here. However, some particular remarks on the compilation from SIL to C^{int} are necessary:

Compiling SIL to C^{int} essentially is the data and operation refinement step which implements ComLisp s-expression data in the linear memory model of C^{int}. C^{int} programs use a *stack* and a *heap* in order to store s-expression representations. SIL operators become C^{int} procedures. The set of operator implementations is called the *core runtimesystem* (chapter B). This constant set of C^{int} modules becomes part of the generated result. Its correctness will not be proved here. Data and operation refinement proofs and in particular a garbage collector correctness proof can be found in [Mic98].

The SIL to C^{int} compiler also includes a set of ComLisp procedures and functions which construct the initial stack and heap segments for global variables, symbols and fat program constants like constant strings and lists. The correctness of these procedures is not proved here as well. Instead, correctness of the core runtimesystem and the generation of the initial heap and stack segments will be handled in a separate report. This report will link the implicit specification given in [GH98b] and the concrete code described here to the mathematical proof given in [Mic98].

4.1 SIL–Compiler: Code Review

Name: CS

Part of: SIL to C^{int}–compiler

Specification:

$$\mathcal{CS}[\![d\ \delta_1\ \delta_2\ \ldots\ \delta_n]\!] \supseteq d'\ datainit\ crts\ \mathcal{CS}_{\mathrm{def}}[\![\delta_1]\!]\zeta\ \mathcal{CS}_{\mathrm{def}}[\![\delta_2]\!]\zeta\ \ldots\ \mathcal{CS}_{\mathrm{def}}[\![\delta_n]\!]\zeta$$

where $d \in$ <`declarations`>$_{\mathrm{SIL}}$, $\delta_1,\ \delta_2,\ \ldots,\ \delta_n \in$ <`definition`>$_{\mathrm{SIL}}$.

The resulting C^{int}–declarations d', the initial stack and heap content *datainit*, the core runtime system *crts* which implements SIL–standard operators, and the heap environment ζ, which maps s-expressions to heap array indices, are defined by $\mathcal{CS}_{\mathrm{decl}}[\![d]\!] \supseteq <\zeta,\ d',\ datainit,\ crts>$.

$\mathcal{CS}.1$

Source–Code:

```
(defun CS (p)
  (let* ((decls (car p))
         (defs (cdr p))
         (env–decl–defs (CSdecl decls))
         (env (car env–decl–defs))
         (decl–defs (cdr env–decl–defs)))
    (append  decl–defs (CSdefs defs env))))
```

Name: CSdecl

Part of: SIL to C^{int}–compiler

Specification:

$$\mathcal{CS}_{\text{decl}}[\![\,(nvars\ funs\ syms\ qc)\,]\!] \supseteq \langle\zeta,\ (funs'\ |s|\ |h|),\ (\texttt{\_DEFDATA}\ s\ h),\ crts\rangle$$

---

where the declarations are from a wellformed program, and $nvars \in \mathbb{N}_0$, $funs, syms \in$ <symbol>*. $qc \in$ <sexpr>* is the list of fat quote constants. $funs'$ is the concatenation of $funs$ and the list of function names defined in $crts$, and we require s, h and the global heap environment ζ to correctly represent $syms$ and qc w.r.t. $nvars$. $crts$ is the core runtime system.

$\mathcal{CS}.2$

Source–Code:

```
(defun CSdecl (decls)
  (let ((nglobals (car decls))
        (funs (append
               (cadr decls)
               (list
               'intern
               '_copy-string
               '_string-member
               '_string=
               'symbol-name
               '_NILname
               '_Tname
               'aref
               'coerce
               'length
               'cdr
               'car
               'cons
               'peek-char
               'write-char
               'read-char
               'char-code
               'code-char
               'mod
               'floor
               '*
               '-
               '+
```

```
            'eql
            '>=
            '<
            'stringp
            'consp
            'characterp
            'integerp
            'symbolp
            'null
            '_aref-error
            '_cc-error
            '_eql-error
            '_assert-string
            '_assert-integer
            '_assert-cons
            '_assert-char
            '_truth
            '_false
            '_true
            '_err
            'abort
            '_COLLECT-CONS
            '_COLLECT-STRING
            '_COLLECT
            '_COLLECT-GARBAGE
            )))
    (syms (caddr decls))
    (s-exprs (cadddr decls)))
(let* ((h (construct-heap syms s-exprs))
       (symbols-tag (car h))
       (symbols-value (cadr h))
       (henv (caddr h))
       (heap (cadddr h))
       (n 1)
       (stack NIL)
       (crts (read)))
  (do ()
      ((< nglobals n))
    (setq stack (list* 0 0 stack))
    (setq n (+ n 1)))
  (setq stack (list* symbols-tag symbols-value stack))
  (list* henv
         (list funs (list-length stack) (list-length heap))
         (list '_DEFDATA stack heap)
         crts))))
```

Comments:

This function uses `(read)` in order to input the core runtimesystem (cf. chapter B). It also uses the function `construct-heap` in order to construct the initial heap from the list of symbols `syms` and the list of fat program constants `s-exprs`. A loop is used in order to construct the initial stack segment `stack` which includes a reference to the symbol table at position 0 and `nglobals` references to `NIL` for the global SIL

variables. That this construction is in conformance with the implicit specification of s and h to *correctly represent syms* and qc w.r.t. *nvars* will be proved in a separate report (cf. our remark at the beginning of this chapter).

Name: CSdefs

Part of: SIL to C^{int}–compiler

Source–Code:

```
(defun CSdefs (defs henv)
  (let* ((result NIL))
    (do ()
     ((null defs))
     (setq result (cons (CSdef (car defs) henv) result))
     (setq defs (cdr defs)))
    (reverse result)))
```

Comments:

This function iterates the function `CSdef` over a list of SIL definitions.

Name: CSdef

Part of: SIL to C^{int}–compiler

Specification:

$$\mathcal{CS}_{\text{def}}[\![\,\texttt{(DEFUN}\ p\ f_1\ \ldots\ f_m\texttt{)}\,]\!]\zeta\ \supseteq\ \texttt{(DEFUN}\ p\ \texttt{(}s\texttt{)}\ \mathcal{CS}_{\text{form}}[\![\,f_1\,]\!]\zeta\ \ldots\ \mathcal{CS}_{\text{form}}[\![\,f_m\,]\!]\zeta\texttt{)}$$

---

where $p \in$ <symbol>, $f_1, \ldots, f_m \in$ <form>$_{\text{SIL}}$,
and $s\ =\ 2 \cdot (\mathit{maxindex}(\,f_1\ \ldots\ f_m\,) + 1)$ (which is 2 for $m = 0$).

$\mathcal{CS}.3$

Source–Code:

```
(defun CSdef (d henv)
  (let* ((p   (cadr d))
         (fs  (cddr d))
         (s (* 2 (+ (max-local-index fs) 1))))
    (list* 'DEFUN p (list s) (CSforms fs henv))))
```

Name: max-local-index

Part of: SIL to C^{int}–compiler

Specification:

$$
\begin{aligned}
maxindex(f_1 \ldots f_n) &:= \max_{j=1,\ldots,n} \{maxindex(f_j)\} \\
maxindex() &:= 0 \\
maxindex(\texttt{(\_COPYC } s\ i\texttt{)}) &:= i \\
maxindex(\texttt{(\_COPY } i_1\ i_2\texttt{)}) &:= \max\{i_1, i_2\} \\
maxindex(\texttt{(\_GCOPY } k\ i\texttt{)}) &:= i \\
maxindex(\texttt{(\_COPYG } i\ k\texttt{)}) &:= i \\
maxindex(\texttt{(}p\ i\texttt{)}) &:= i \\
maxindex(\texttt{(LIST*} n\ i\texttt{)}) &:= i + n \\
maxindex(\texttt{(IF } i\ \texttt{(}f_1 \ldots f_n\texttt{)}\ \texttt{(}f_{n+1} \ldots f_m\texttt{))}) &:= \max\{i, \max_{j=1,\ldots,m} \{maxindex(f_j)\}\} \\
maxindex(\texttt{(DO } i\ \texttt{(}f_1\texttt{)}\ f_2 \ldots f_n\texttt{)}) &:= \max\{i, \max_{j=1,\ldots,n} \{maxindex(f_j)\}\}
\end{aligned}
$$

Source–Code:

```
(defun max–local–index (forms)
  (if (null forms)
      0
      (max (max–local–index–form (car forms)) (max–local–index (cdr forms)))))


(defun max–local–index–form (form)
  (cond
    ((consp form)
     (let ((key (car form))
           (args (cdr form)))
       (cond
         ((eql key 'IF)
          (max (max–local–index (cadr args))
               (max–local–index (caddr args))))
         ((eql key '_COPY) (max (car args) (cadr args)))
         ((eql key '_COPYG) (car args))
         ((eql key '_GCOPY) (cadr args))
         ((eql key '_COPYC) (cadr args))
         ((eql key 'DO) (max
                         (max–local–index (cadr args))
                         (max–local–index (cddr args))))
         ((eql key 'LIST*)
          (– (+ (cadr args) (car args)) 1))
         (T 0))))
    (T 0)))
```

Comments:

Given a SIL–form f, $maxindex(f)$ is the largest local stack index i used in f.

Name: CSforms

Part of: SIL to C^{int}–compiler

Source–Code:

```
(defun CSforms (forms henv)
  (if (null forms)
      NIL
      (append
       (CSform (car forms) henv)
       (CSforms (cdr forms) henv))))
```

Issues:

This function implements the implicit linear recursion, that appears while specifying the transformation of sequences of SIL forms

$$f_1 \ \ldots \ f_n \quad \text{into sequences} \quad \mathcal{CS}_{\text{form}}[\![f_1]\!]\zeta \ \ldots \ \mathcal{CS}_{\text{form}}[\![f_n]\!]\zeta \,.$$

of compiled forms. Sequences are implemented as lists, and the resulting target code lists have to be appended. `henv` holds the heap environment representation of ζ.

Name: CSform

Part of: SIL to C^{int}–compiler

Source–Code:

```
(defun CSform (form henv)
  (cond
   ((consp form)
    (let ((key (car form))
          (args (cdr form)))
      (cond
       ;; IF-------------------------------------------------------------------
       ((eql key 'IF) ... )

       ;; DO-------------------------------------------------------------------
       ((eql key 'DO) ... )

       ;; _COPY----------------------------------------------------------------
       ((eql key '_COPY) ... )

       ;; _COPYG---------------------------------------------------------------
       ((eql key '_COPYG) ... )

       ;; _GCOPY---------------------------------------------------------------
       ((eql key '_GCOPY) ... )

       ;; _COPYC---------------------------------------------------------------
       ((eql key '_COPYC) ... )

       ;; LIST*----------------------------------------------------------------
       ((eql key 'LIST*) ...)

       ;; application----------------------------------------------------------
       (T ... ))))

   (T (errorstop 21))))
```

Name: CSform translation of procedure calls

Part of: SIL to C^{int}–compiler

Specification:

$$\mathcal{CS}_{\text{form}}[\![(p\ i)]\!]\zeta \quad \supseteq \quad (p\ 2i)$$

where $p \in$ <symbol>

$\mathcal{CS}.4$

Source–Code:

```
(defun CSform (form henv)
  (cond
   ((consp form)
    (let ((key (car form))
          (args (cdr form)))
      (cond
        ...
        ;; application------------------------------------------------------------
        (T
         (list (list key (* 2 (car args))))))))
   ...
   ))
```

Comments:

Note that this is the final case in the definition of `CSform` on page 57. Every form, which is a list beginning with a non-keyword, is handled as a procedure call.

Related Documents: Definition of `CSform` on page 57

Name: CSform _COPYG, _GCOPY, _COPY–translation

Part of: SIL to C^{int}–compiler

Specification:

$$\begin{array}{rcl}
\mathcal{CS}_{\text{form}}[\![\texttt{(_COPY } i\ j\texttt{)}]\!]\zeta & \supseteq & \texttt{(_SETLOCAL (_LOCAL } 2i\texttt{) } 2j\texttt{)} \\
 & & \texttt{(_SETLOCAL (_LOCAL } 2i{+}1\texttt{) } 2j{+}1\texttt{)} \\
\mathcal{CS}_{\text{form}}[\![\texttt{(_GCOPY } i\ j\texttt{)}]\!]\zeta & \supseteq & \texttt{(_SETLOCAL (_STACK } 2i\texttt{) } 2j\texttt{)} \\
 & & \texttt{(_SETLOCAL (_STACK } 2i{+}1\texttt{) } 2j{+}1\texttt{)} \\
\mathcal{CS}_{\text{form}}[\![\texttt{(_COPYG } i\ j\texttt{)}]\!]\zeta & \supseteq & \texttt{(_SETSTACK (_LOCAL } 2i\texttt{) } 2j\texttt{)} \\
 & & \texttt{(_SETSTACK (_LOCAL } 2i{+}1\texttt{) } 2j{+}1\texttt{)}
\end{array}$$

where for _GCOPY and _COPYG we require $i \geq 1$.

$\mathcal{CS}.5$

Source–Code:

```
(defun CSform (form henv)
  (cond
    ((consp form)
     (let ((key (car form))
           (args (cdr form)))
       (cond
         ...
         ;; _COPY-------------------------------------------------------------
         ((eql key '_COPY)
          ;; (_COPY i j)  --> (SETLOCAL (LOCAL 2i) 2j)   (... (... 2i+1) 2j+1)
          (let ((i (car args))
                (j (cadr args)))
            (list
             (list '_SETLOCAL (list '_LOCAL (* 2 i)) (* 2 j))
             (list '_SETLOCAL (list '_LOCAL (+ (* 2 i) 1)) (+ (* 2 j) 1)))))

         ;; _COPYG------------------------------------------------------------
         ((eql key '_COPYG)
          ;; (_COPYG i j)  --> (SETSTACK (LOCAL 2i) 2j)  (... (... 2i+1) 2j+1)
          (let ((i (car args))
                (j (cadr args)))
            (list
             (list '_SETSTACK (list '_LOCAL (* 2 i)) (* 2 j))
             (list '_SETSTACK (list '_LOCAL (+ (* 2 i) 1)) (+ (* 2 j) 1)))))

         ;; _GCOPY------------------------------------------------------------
         ((eql key '_GCOPY)
```

```
         ;; (_GCOPY i j)  --> (SETLOCAL (STACK 2i) 2j)  (... (... 2i+1) 2j+1)
         (let ((i (car args))
               (j (cadr args)))
           (list
            (list '_SETLOCAL (list '_STACK (* 2 i)) (* 2 j))
            (list '_SETLOCAL (list '_STACK (+ (* 2 i) 1)) (+ (* 2 j) 1)))))

        ...
        )))
    ...
    ))
```

Related Documents: Definition of CSform on page 57

Name: CSform DO–translation

Part of: SIL to C^{int}–compiler

Specification:

$$\mathcal{CS}_{\text{form}}[\![\,(\texttt{DO}\ i\ (f_1 \ldots f_m)\ f_{m+1}\ \ldots\ f_n)\,]\!]\zeta \supseteq (\texttt{DO}\ (\texttt{PROGN}\ \mathcal{CS}_{\text{form}}[\![\,f_1\,]\!]\zeta \ \ldots\ \mathcal{CS}_{\text{form}}[\![\,f_m\,]\!]\zeta)\ (\texttt{\_!=}\ (\texttt{\_LOCAL}\ 2i)\ \textit{niltag})\ (\texttt{PROGN}\ \mathcal{CS}_{\text{form}}[\![\,f_{m+1}\,]\!]\zeta \ \ldots\ \mathcal{CS}_{\text{form}}[\![\,f_n\,]\!]\zeta))$$

where $f_1, \ldots, f_n \in \texttt{<form>}_{\text{SIL}}$.

$\mathcal{CS}.6$

Source–Code:

```
(defun CSform (form henv)
  (cond
   ((consp form)
    (let ((key (car form))
          (args (cdr form)))
      (cond
        ...
        ;; DO----------------------------------------------------------------------
        ((eql key 'DO)
         ;; (DO k (pred) body) ->
         ;;     (DO (progn pred') (_!= (LOCAL 2k) niltag) (progn body'))
         (let ((k (car args))
               (pred (cadr args))
               (body (cddr args)))
           (list
            (list
             'DO
             (cons 'PROGN (CSforms pred henv))
             (list '_!= (list '_LOCAL (* 2 k)) 0) ; 0 = <nil-tag>
             (cons 'PROGN (CSforms body henv))))))
        ...
        )))
   ...
   ))
```

Related Documents: Definition of CSform on page 57

Name: CSform IF–translation

Part of: SIL to C^{int}–compiler

Specification:

$$\mathcal{CS}_{\text{form}}[\![\texttt{(IF } i \texttt{ (}f_1 \ \ldots \ f_n\texttt{) (}f_{n+1} \ \ldots \ f_m\texttt{))}]\!]\zeta \quad \supseteq$$
$$\texttt{(IF (\_!= (\_LOCAL } 2i\texttt{) } \mathit{niltag}\texttt{)}$$
$$\texttt{(PROGN } \mathcal{CS}_{\text{form}}[\![f_1]\!]\zeta \ \ldots \ \mathcal{CS}_{\text{form}}[\![f_n]\!]\zeta\texttt{)}$$
$$\texttt{(PROGN } \mathcal{CS}_{\text{form}}[\![f_{n+1}]\!]\zeta \ \ldots \ \mathcal{CS}_{\text{form}}[\![f_m]\!]\zeta\texttt{))}$$

where $f_1, \ldots, f_m \in$ <form>$_{\text{SIL}}$.

$\mathcal{CS}.7$

Source–Code:

```
(defun CSform (form henv)
  (cond
   ((consp form)
    (let ((key (car form))
          (args (cdr form)))
      (cond
       ;; IF------------------------------------------------------------------
       ((eql key 'IF)
        ;; (IF n (then) (else)) --> (IF n'!=0 (progn then') (progn else'))
        (let ((i (car args)))
          (list
           (list
            'IF (list '_!= (list 'LOCAL (* 2 i)) 0)     ; 0 = <nil-tag>
            (cons 'PROGN (CSforms (cadr args) henv))
            (cons 'PROGN (CSforms (caddr args) henv))))))
        ...
        )))
        ...))
```

Related Documents: Definition of CSform on page 57

Name: CSform LIST*–translation

Part of: SIL to C^{int}–compiler

Specification:

$$\mathcal{CS}_{\text{form}}[\![\,\texttt{(LIST*}\ i\ n\texttt{)}\,]\!]\zeta \ \supseteq\ \begin{array}{l}\texttt{(CONS}\ m\texttt{)}\\ \texttt{(CONS}\ m\text{–}2\texttt{)}\\ \ldots\\ \texttt{(CONS}\ 2i{+}2\texttt{)}\\ \texttt{(CONS}\ 2i\texttt{)}\end{array}$$

where $m = 2(i + n - 2)$, arity $n \geq 1$ and stack index $i \geq 0$.

$\mathcal{CS}.8$

Source–Code:

```
(defun CSform (form henv)
  (cond
   ((consp form)
    (let ((key (car form))
          (args (cdr form)))
      (cond
        ...
        ;; LIST*
        ((eql key 'LIST*)
         (let* ((n (car args))
                (i (cadr args))
                (m (* 2 (- (+ i n) 2))))
           (CSlist* m (* 2 i))))
        ...
        )))
   ...
   ))
```

Source–Code:

```
(defun CSlist* (b e)
  (if (< b e)
      NIL
      (cons
       (list 'CONS b)
       (CSlist* (- b 2) e))))
```

Related Documents: Definition of CSform on page 57

Name: CSform –translation

Part of: SIL to C^{int}–compiler

Specification:

$$\mathcal{CS}_{\text{form}}[\![\,\texttt{(\_COPYC}\ n\ i\texttt{)}\,]\!]\zeta \supseteq \texttt{(\_SETLOCAL}\ \textit{numbertag}\ 2i\texttt{)}\ \texttt{(\_SETLOCAL}\ n\ 2i{+}1\texttt{)}$$

$$\mathcal{CS}_{\text{form}}[\![\,\texttt{(\_COPYC}\ c\ i\texttt{)}\,]\!]\zeta \supseteq \texttt{(\_SETLOCAL}\ \textit{charactertag}\ 2i\texttt{)}\ \texttt{(\_SETLOCAL}\ \text{ord}(c)\ 2i{+}1\texttt{)}$$

$$\mathcal{CS}_{\text{form}}[\![\,\texttt{(\_COPYC T}\ i\texttt{)}\,]\!]\zeta \supseteq \texttt{(\_SETLOCAL}\ \textit{t-tag}\ 2i\texttt{)}\ \texttt{(\_SETLOCAL}\ \textit{t-value}\ 2i{+}1\texttt{)}$$

$$\mathcal{CS}_{\text{form}}[\![\,\texttt{(\_COPYC NIL}\ i\texttt{)}\,]\!]\zeta \supseteq \texttt{(\_SETLOCAL}\ \textit{niltag}\ 2i\texttt{)}\ \texttt{(\_SETLOCAL}\ \textit{nilvalue}\ 2i{+}1\texttt{)}$$

where $n \in$ <integer> and $c \in$ <character>.

$\mathcal{CS}.9$

$$\mathcal{CS}_{\text{form}}[\![\,\texttt{(\_COPYC}\ s\ i\texttt{)}\,]\!]\zeta \supseteq \texttt{(\_SETLOCAL}\ \textit{tag}\ 2i\texttt{)}\ \texttt{(\_SETLOCAL}\ \textit{value}\ 2i{+}1\texttt{)}$$

where $(tag, value) = \zeta(s)$.

$\mathcal{CS}.10$

Source–Code:

```
(defun CSform (form henv)
  (cond
   ((consp form)
    (let ((key (car form))
          (args (cdr form)))
      (cond
        ...
        ((eql key '_COPYC)
         ;; (_COPYC sexpr j)  --> (SETLOCAL tag 2j)  (SETLOCAL value 2j+1)
         (let* ((sexpr (car args))
```

```
            (j (cadr args)))
          (cond
            ((integerp sexpr)
             (list
              (list '_SETLOCAL 3 (* 2 j)) ; 3 = <integer-tag>
              (list '_SETLOCAL sexpr (+ 1 (* 2 j)))))
            ((characterp sexpr)
             (list
              (list '_SETLOCAL 4 (* 2 j)) ; 4 = <character-tag>
              (list '_SETLOCAL (char-code sexpr) (+ 1 (* 2 j)))))
            ((eql sexpr T)
             (list
              (list '_SETLOCAL 1 (* 2 j)) ; 1 = <t-tag>
              (list '_SETLOCAL 1 (+ 1 (* 2 j)))) ; 1 = <t-value>
            ((null sexpr)
             (list
              (list '_SETLOCAL 0 (* 2 j)) ; 0 = <t-tag>
              (list '_SETLOCAL 0 (+ 1 (* 2 j)))) ; 0 = <t-value>
            (T
             (let ((rep (assoc-equal sexpr henv)))
               (if (null rep)
                   (errorstop 51))
               (let ((tag (cadr rep))
                     (value (cddr rep)))
                 (list
                  (list '_SETLOCAL tag (* 2 j))
                  (list '_SETLOCAL value (+ 1 (* 2 j))))))))
      ...
      )))
..
))
```

Related Documents: Definition of `CSform` on page 57

Name: construct-heap

Part of: SIL to C^{int}–compiler

Specification:

The heap construction is implicitly specified in the specification of $\mathcal{CS}_{\text{decl}}$.

Source–Code:

```
(defun construct–heap (syms s–exprs)
  (let* ((symlist (construct–symbol–list syms NIL (empty–queue)))
         (tag (car symlist))
         (value (cadr symlist))
         (henv (caddr symlist))
         (heap (cadddr symlist)))
    (list* tag value (construct–s–exprs s–exprs henv heap))))

(defun construct–s–exprs (s–exprs henv heap)
  (do ()
   ((null s–exprs))
   (let ((s (construct (car s–exprs) henv heap)))
     (setq henv (caddr s))
     (setq heap (cadddr s)))
   (setq s–exprs (cdr s–exprs)))
  (list henv (queue2list heap)))

(defun construct (s henv heap)
  ;; –> (tag value henv' heap')
  (let ((rep NIL))
    (cond
      ((integerp s)
       (list (heap–tag s) s henv heap))
      ((characterp s)
       (list (heap–tag s) (char–code s) henv heap))
      ((null s)
       (list (heap–tag s) 0 henv heap)) ; 0 = <nil–value>
      ((eql s T)
       (list (heap–tag s) 1 henv heap)) ; 1 = <t–value>
      ((setq rep (assoc–equal s henv))
       (list (cadr rep) (cddr rep) henv heap))
      ((stringp s) (construct–string s henv heap))
      ((consp s)  (construct–cons s henv heap))
      (T (errorstop 22)))))

(defun heap–addr (heap) (queue–length heap))

(defun construct–cons (c henv heap)
  (let* ((a (construct (car c) henv heap))
         (a–tag (car a))
         (a–value (cadr a))
         (a–henv (caddr a))
         (a–heap (cadddr a))
         (d (construct (cdr c) a–henv a–heap))
         (d–tag (car d))
```

```
        (d-value (cadr d))
        (d-henv (caddr d))
        (d-heap (cadddr d))
        (ha (heap-addr d-heap)))
    (list
     (heap-tag c)
     ha
     (list* (cons c (cons (heap-tag c) ha)) d-henv)
     (put-queue-list (list a-tag a-value d-tag d-value) d-heap))))

(defun heap-tag (s)
  (cond
    ((null s) 0)                        ; 0 = <nil-tag>
    ((eql s T) 1)                       ; 1 = <t-tag>
    ((symbolp s) 2)                     ; 2 = <symbol-tag>
    ((integerp s) 3)                    ; 3 = <integer-tag>
    ((characterp s) 4)                  ; 4 = <character-tag>
    ((consp s) 5)                       ; 5 = <cons-tag>
    ((stringp s) 6)                     ; 6 = <string-tag>
    (T (errorstop 23))))

(defun construct-symbol-list (symlist henv heap)
  (cond
    ((null symlist)
     (list (heap-tag symlist) 0 henv heap)) ; 0 = <nil-value>
    (T
     (let* ((sl (construct-symbol-list (cdr symlist) henv heap))
            (tag (car sl))
            (value (cadr sl)))
       (setq henv (caddr sl))
       (setq heap (cadddr sl))
       (let* ((symbol (car symlist))
              (sym (construct-string (symbol-name symbol) henv heap))
              (symtag (car sym))
              (symvalue (cadr sym)))
         (setq henv (caddr sym))
         (setq heap (cadddr sym))
         (let ((ha (heap-addr heap)))
           (list (heap-tag (cons nil nil))
                 ha
                 (list* (cons symbol (cons (heap-tag symbol) ha)) henv)
                 (put-queue-list (list symtag symvalue tag value) heap))))))))

(defun construct-string (a henv heap)
  (let* ((i 0)
         (rep (put-queue-list (list (heap-tag (length a)) (length a))
                              (empty-queue))))
    (do ()
     ((null (< i (length a))))
     (let* ((r (construct (aref a i) henv heap))
            (tag (car r))
            (value (cadr r)))
       (setq henv (caddr r))
```

```
      (setq heap (cadddr r))
      (setq rep (put-queue-list (list tag value) rep)))
    (setq i (+ 1 i)))
  (let* ((ha (heap-addr heap)))
    (list
     (heap-tag a)
     ha
     (list* (cons a (cons (heap-tag a) ha)) henv)
     (setq heap (put-queue-list (queue2list rep) heap))))))
```

Comments:

Here our remark at the beginning of this chapter applies. The correctness of this compiler part will be proved elsewhere.

Note, however, that the function `construct` does not mention the `symbol` case. It depends on the fact that the symbols and the complete symbol table are constructed first. This is because we want the symbol table to be entirely represented as a dense initial part of the heap. Therefore, for proper SIL programs, every symbol occuring in the fat program constants `s-exprs` has already been represented.

Chapter 5

C^{int} to TASM

The compiling relation between C^{int} and TASM is specified in [GH98b] to be a syntactical mapping between C^{int} source programs and TASM target programs. It is defined recursively using several auxiliary relations which correspond to the C^{int} source program structure. The types are as follows:

$$
\begin{array}{rcl}
\mathcal{CC} & \in & \texttt{<program>}_{C^{int}} \rightharpoonup \texttt{<program>}_{TASM} \\
\mathcal{CC}_{decl} & \in & \texttt{<declarations>}_{C^{int}} \rightharpoonup (\, globenv \times \texttt{<decls>}_{TASM} \times \texttt{<module>}_{TASM}{}^{*}\,) \\
\mathcal{CC}_{data} & \in & \texttt{<data>}_{C^{int}} \rightharpoonup \texttt{<data-module>}_{TASM}{}^{*}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{CC}_{def} & \in & \texttt{<definition>}_{C^{int}} \times globenv \rightharpoonup \texttt{<code-module>}_{TASM} \\
\mathcal{CC}_{stmt} & \in & \texttt{<statement>}_{C^{int}} \times globenv \times \mathbb{N}_0 \rightharpoonup \texttt{<op>}_{TASM}{}^{*} \\
\mathcal{CC}_{expr} & \in & \texttt{<expression>}_{C^{int}} \times globenv \times \mathbb{N}_0 \rightharpoonup \texttt{<op>}_{TASM}{}^{*}
\end{array}
$$

where

$$
\begin{array}{rcl}
globenv & = & procenv \times \mathbb{N}_0 \times \mathbb{N}_0 \\
procenv & = & \texttt{<ident>}_{C^{int}} \xrightarrow{\text{part}} \mathbb{N}_0
\end{array}
$$

Like before these relations are now compared to ComLisp implementations (cf. page 17).

5.1 Lisp Data Representation

The compiling specification for the translation from C^{int} to TASM uses a compile time environment to map procedure identifiers to indices of a subroutine jump table, which will be built by the bootstrap loader. This compile time environment is implemented as a list of procedure identifiers, where the position of an identifier in this list determines its index. (See also page 18 for further data representations).

5.2 Separation of machine independent and dependent parts

The implementation of the C^{int} to TASM compiling specification is seperated in a machine independent frontend which performs syntactical decomposition of SIL–programs and a backend which generates machine dependent code. This separation simplifies the construction of code fopr other processors because only the code generator functions in the backend have to be replaced. The names of the machine dependend code generator functions start with `CG-` or `TC-`.

5.3 C^{int}–Compiler: Code Review

Name: Translation of C^{int}-programs

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}[\![d\ \delta\ p_1\ p_2\ \ldots\ p_n]\!] \supseteq \begin{array}{l} d'\ \textit{init-code} \\ \mathcal{CC}_{\text{data}}[\![\delta]\!] \\ \mathcal{CC}_{\text{def}}[\![p_1]\!]\,\varphi \\ \mathcal{CC}_{\text{def}}[\![p_2]\!]\,\varphi \\ \ldots \\ \mathcal{CC}_{\text{def}}[\![p_n]\!]\,\varphi \end{array}$$

where $d \in$ <declarations>$_{C^{int}}$, $\delta \in$ <data-definition>, p_1, p_2, ... $p_n \in$ <definition>$_{C^{int}}$ and the sequence $d\ \delta\ p_1\ p_2\ \ldots\ p_n$ forms a wellformed C^{int}–program.

The TASM–declarations d', initialization code *init-code*, and the global environment φ are results of compiling the declarations, i.e. $\mathcal{CC}_{\text{decl}}[\![d]\!] \supseteq <\varphi, d', \textit{init-code}>$.

$\mathcal{CC}.1$

Source–Code:

```
(defun CC (p)
  (let* ((cintdecl (car p))
         (cintdata (cadr p))
         (cintdefs (cddr p))
         (cintenv–asmdecl–asmdefs (CCdecl cintdecl))
         (cintenv (car cintenv–asmdecl–asmdefs)))
    (append (cdr cintenv–asmdecl–asmdefs)
            (append (CCdata cintdata cintenv)
                    (CCdefs cintdefs cintenv)))))
```

Comments:

`CCdata` is called with the compile time environment as an additional parameter which however is ignored by `CCdata`.

Related Documents: The definition of `CCdefs` on page 75
The definition of `CCdata` on page 78
The definition of `CCdecl` on the next page

Name: Translation of declarations

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{decl}}[\![\,(\mathit{funs}\ \mathit{s\text{-}size}\ \mathit{h\text{-}size})\,]\!] \ \supseteq\ <\varphi,\ (3+|\mathsf{dom}\ \varphi|)\ ,\ \mathit{init\text{-}code}>$$

where $\mathit{funs} \in$ <ident>*, em s-size, $\mathit{h\text{-}size} \in$ <integer>, and the declarations ($\mathit{syms}\ \mathit{s\text{-}size}\ \mathit{h\text{-}size}$) are from a wellformed C^{int}–program.

The global environment $\varphi = <\psi, \mathit{s\text{-}size}, \mathit{h\text{-}size}> \in \mathit{globenv}$ and the procedure environment ψ must be injective.

$\mathcal{CC}.2$

Source–Code:

```
(defun CCdecl (decls)
  (cons decls (CG–initialize '_MAIN decls)))
```

Source–Code:

```
(defun CG–initialize (main genv)
  (let* ((funs (car genv))
         (s–size (cadr genv))
         (h–size (caddr genv))
         (cint–init–code
          (list
           ;; startaddress of jump–table is in workspace address 0
           ;; and memtop in workspace address 1

           ;; round memtop down to nearest word address
           'LDL 1
           'OPR 63                      ; WCNT
           'OPR 52                      ; BCNT

           'LDNLP –2                    ; leave enough space for a single call

           'STL 4                       ; 4 = <memtop>
           'LDLP 13                     ; 13 = <stack>
           'ADC (* 100 1024)            ; kB stack
           'STL 1                       ; 1 = <heap>
           'LDL 1                       ; 1 = <heap>
           'LDL 4                       ; 4 = <memtop>
           'ADC (– 0 (* 100 1024))      ; kB returnstack
           'STL 6                       ; 6 = <rstack>
           'LDL 6                       ; 6 = <rstack>
           'STL 5                       ; 5 = <rp>

           ;; setup <base>
```

```
'LDC s-size
'STL 11                         ; 11 = <temp>

'LDL 11                         ; 11 = <temp>
'LDLP 13                        ; 13 = <stack>
'OPR 10                         ; WSUB
'STL 3                          ; 3 = <base>
'LDL 3                          ; 3 = <base>
'LDL 1                          ; 1 = <heap>
'OPR 9                          ; GT
'CJ 2
'OPR 16                         ; SETERR

'LDC -1                         ; initialize character buffer
'STL 12                         ; 12 = <lastch>

;; move stack-image
'LDL 11                         ; 11 = <temp>      ; counted to zero?
'CJ  15

'LDL 11                         ; 11 = <temp>      ; decrement counter
'ADC -1
'STL 11                         ; 11 = <temp>

'LDL 11                         ; 11 = <temp>
'LDL 0                          ; 0 = <start>
'LDNL 1                         ; start of stack-data-area
'OPR 10                         ; WSUB
'LDNL 0
'LDL 11                         ; 11 = <temp>
'LDLP 13                        ; 13 = <stack>
'OPR 10                         ; WSUB
'STNL 0

'J   -17                        ; jump to condition

;; setup <quotetop> and <heaptop>
'LDC h-size
'STL 11                         ; 11 = <temp>

'LDL 11                         ; 11 = <temp>
'STL 10                         ; 10 = <quotetop>
'LDL 10                         ; 10 = <quotetop>
'STL 2                          ; 2 = <heaptop>
'LDL 2                          ; 2 = <heaptop>
'LDL 1                          ; 1 = <heap>
'OPR 10                         ; WSUB
'LDL 6                          ; 6 = <rstack>
'OPR 9                          ; GT
'CJ 2
'OPR 16                         ; SETERR

;; move heap-image
```

```
        'LDL 11                         ; 11 = <temp>      ; counted to zero?
        'CJ  15

        'LDL 11                         ; 11 = <temp>      ; decrement counter
        'ADC -1
        'STL 11                         ; 11 = <temp>

        'LDL 11                         ; 11 = <temp>
        'LDL 0                          ; 0 = <start>
        'LDNL 2                         ; start of heap-data-area
        'OPR 10                         ; WSUB
        'LDNL 0
        'LDL 11                         ; 11 = <temp>
        'LDL 1                          ; 1 = <heap>
        'OPR 10                         ; WSUB
        'STNL 0

        'J   -17                        ; jump to condition

        ;; setup channels
        'OPR 66                         ; MINT
        'STL 8                          ; 8 = <outchan>
        'LDL 8                          ; 8 = <outchan>
        'LDNLP 4
        'STL 9                          ; 9 = <inchan>

        ;; call main function
        'LDC 0
        'LDL 0                          ; 0 = <start>
        'LDNL (TC-function-index main funs) ; adjust index
        'OPR 6                          ; GCALL

        ;; terminate succesfully
        'OPR 16                         ; SETERR
        )))
    (list
     (list (+ 3 (list-length funs)))
     (list* '_DEFCODE '_CINT-TASM-INIT 0 cint-init-code))))
```

Issues:

Construct the global compile time environment, the TASM–declaration and the C^{int}-initialization code.

The compile time environment consists of a triple with the following components:

1. the procedure environment which maps identifiers to jump table indices,
2. the size of the initial stack
3. the size of the initial heap

Comments:

`CCdecl` uses the auxiliary function `CG-initialize` in order to separate machine independent and machine dependent parts of the C^{int}–compiler.

Name: Translation of procedure definition sequences

Part of: C^{int} to TASM–compiler

Source–Code:

```
(defun CCdefs (defs genv)
  (let ((result NIL))
    (do ()
     ((null defs))
     (setq result (rappend (CCdef (car defs) genv) result))
     (setq defs (cdr defs)))
    (reverse result)))
```

Issues:

This function implements the implicit linear recursion, that appears in the specification of the transformation of sequences of C^{int} definitions

$$p_1 \ \ldots \ p_n \quad \text{to sequences} \quad \mathcal{CC}_{\text{def}}[\![p_1]\!]\varphi \ \ldots \ \mathcal{CC}_{\text{def}}[\![p_n]\!]\varphi\,.$$

of compiled definitions. Sequences are implemented as lists, and `genv` holds the representation of the global environment φ.

Comments:

See remark for `CLdefs` on page 22.

Name: TC-check-word-size

Part of: C^{int} to TASM–compiler

Specification:

No explicit specification. Constants in generated TASM–programs must respect target machine resource restrictions.

Source–Code:

```
(defvar *Bytes–per–Word*) (setq *Bytes–per–Word* 4)
```

Comments:

The target machine in sight is a 32bit transputer, thus $BytesPerWord = 4$. The transputer uses 2's complement number representation.

Source–Code:

```
(defvar *MIN–INT*) (setq *MIN–INT* (let ((n –1) (i 1))
                                     (do ()
                                         ((>= i (* 8 *Bytes–per–Word*)))
                                       (setq n (* 2 n))
                                       (setq i (+ 1 i)))
                                     n))
```

Comments:

This calculates *minint* to be $-(2^{8*BytesPerWord-1})$, thus for $BytesPerWord = 4$ we get $minint = -2147483648$.

Source–Code:

```
(defvar *MAX–INT*) (setq *MAX–INT* (– –1 *MIN–INT*))
```

Comments:

maxint is the 1's complement of *minint*. For $BytesPerWord = 4$ we get $maxint = 2147483647$.

Source–Code:

```
(defun TC–check–word–size (l)
  (cond
    ((consp l)
     (do ()
         ((null (and l (TC–check–word–size (car l)))))
       (setq l (cdr l)))
     (if l NIL T))
    ((integerp l)
     (and
      (>= *MAX–INT* l)
      (>= l *MIN–INT*)))
    (T)))
```

Issues:

Test if any of the constants in list *l* is outside the range of numbers representable on the transputer. If so return `NIL` if all are in range then return `T`.

Comments:

`TC-check-word-size` is used to check constants in TASM–data and –code modules. Code modules may be nested assembly language lists and consequently `TC-check-word-size` does a full (depth first) traversal. Each constant is tested to be in range [*minint*, *maxint*].

Related Documents: definition of `CG-check-data` and `CG-check-code` on pages 78 and 79.

Name: Translation of data definitions

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{data}}[\![\,(\texttt{\_DEFDATA}\ (s)\ (h)\,)\,]\!] \supseteq (\texttt{\_DEFDATA \_STACK}\ 1\ s)\ (\texttt{\_DEFDATA \_HEAP}\ 2\ h)$$

where $s = s_1 \ldots s_n \in \langle\texttt{word}\rangle^*$ and $h = h_1 \ldots h_m \in \langle\texttt{word}\rangle^*$ and $n, m \in \langle\texttt{word}\rangle$.

$\mathcal{CC}.3$

Source–Code:

```
(defun CCdata (definition genv)
  (CG–check–data
   (list
    (CG–data '_STACK (cadr definition) genv)
    (CG–data '_HEAP (caddr definition) genv))))
```

Source–Code:

```
(defun CG–check–data (data)
  (if (TC–check–word–size data)
      data
    (errorstop 35)))
```

Source–Code:

```
(defun CG–data (name data genv)
  (cond
   ((eql name '_STACK)
    (if (null (eql (list–length data) (cadr genv)))
        (errorstop 31))
    (list* '_DEFDATA name 1 data))
   ((eql name '_HEAP)
    (if (null (eql (list–length data) (caddr genv)))
        (errorstop 32))
    (list* '_DEFDATA name 2 data))
   (T
    (errorstop 47))))
```

Comments:

The function `CD-check-data` assures that all data words in the generated data module are within the allowed range for the target machine. Violation causes irregular program termination.

Name: Translation of procedure definitions

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{def}}[\![\,\texttt{(DEFUN } f\ \texttt{(}\sigma\texttt{)}\ s_1\ \ldots\ s_n\texttt{)}\,]\!]\,\varphi \ \supseteq\ \texttt{(\_DEFCODE } f\ \psi(f)\ \texttt{(}\mathit{entrycode}(\sigma)\texttt{)}\ \mathcal{CC}_{\text{stmt}}[\![\,s_1\,]\!]\,\varphi\,\sigma\ \ldots\ \mathcal{CC}_{\text{stmt}}[\![\,s_n\,]\!]\,\varphi\,\sigma\ \texttt{(}\mathit{exitcode}\texttt{))}$$

where $f \in$ <symbol>, $s_1,\ s_2,\ \ldots\ s_n \in$ <statement>$_{C^{\text{int}}}$, $\sigma \in$ <word>, and $\varphi = <\psi,\ \mathit{s\text{-}size},\ \mathit{h\text{-}size}>$.

$\mathcal{CC}.4$

Source–Code:

```
(defun CCdef (definition genv)
  (let* ((name (cadr definition))
         (framesize (caaddr definition))
         (body (cdddr definition)))
    (CG–check–code
     (list (CG–fun name framesize (CCstmts body genv framesize) genv)))))
```

Source–Code:

```
(defun CG–check–code (fun)
  (if (TC–check–word–size fun)
      fun
    (errorstop 34)))
```

Source–Code:

```
(defun CG–fun (name frame–size compiled–body genv)
  (list '_DEFCODE name (TC–function–index name (car genv))
        (CG–fun–entry frame–size)
        compiled–body
        (CG–fun–exit frame–size)))
```

Comments:

Machine independent and machine dependent definitions are separated. The operands in the resulting assembly code are checked for ressource violations by means of calling `CG-check-code`.

Related Documents: The definitions of `CG-fun-entry` and `CG-fun-exit` on pages 81.

Name: Environment lookup

Part of: C^{int} to TASM–compiler

Specification:

No explicit specification

Source–Code:

```
(defun TC–function–index (ident funs)
  (let ((rest (member ident funs)))
    (if rest
        (+ 2 (list–length rest))
      (errorstop 36))))
```

Issues:

Look up procedure identifier in the procedure environment.

Comments:

The procedure environment is implemented as a list of procedure identifiers. The position in this list determines the associated index. Because the indices 0, 1, and 2 are fixed allocated to the C^{int}–initialization code, the data module for the initial stack image and the data module for the initial heap image.

Related Documents: Translation of procedure calls on page 94.

Name: Procedure entry and exit code

Part of: C^{int} to TASM–compiler

Specification:

entrycode(σ) =
```
;; Register A contains return address,
;; Register B contains frame offset
ldl rp   stnl 0              ; save return address on return stack
ldl base ldl rp  stnl 1      ; save frame pointer on return stack
ldl base wsub    stl base    ; adjust frame pointer
ldl rp   ldnlp 2 stl rp      ; adjust return stack pointer
ldl rp   ldl memtop gt       ; check return stack overflow
cj 2     seterr
ldl base ldnlp σ             ; check for stack overflow
ldl heap gt
cj 2     seterr
```

exitcode =
```
ldl rp   ldnlp -2  stl rp    ; adjust return stack pointer
ldl rp   ldnl 1    stl base  ; restore frame pointer
ldl rp   ldnl 0    gcall
```

where $\sigma \in$ <word>.

$\mathcal{CC}.5$

Source–Code:

```
(defun CG-fun-entry (frame)
  (list
   'LDL 5                              ; 5 = <rp>
   'STNL 0
   'LDL 3                              ; 3 = <base>
   'LDL 5                              ; 5 = <rp>
   'STNL 1
   'LDL 3                              ; 3 = <base>
   'OPR 10                             ; WSUB
   'STL 3                              ; 3 = <base>
   'LDL 5                              ; 5 = <rp>
   'LDNLP 2
   'STL 5                              ; 5 = <rp>
   'LDL 5                              ; 5 = <rp>
   'LDL 4                              ; 4 = <memtop>
   'OPR 9                              ; GT
   'CJ 2
   'OPR 16                             ; SETERR
   'LDL 3                              ; 3 = <base>
   'LDNLP frame
   'LDL 1                              ; 1 = <heap>
   'OPR 9                              ; GT
```

```
   'CJ 2
   'OPR 16                              ; SETERR
   ))
```

Source–Code:

```
(defun CG-fun-exit (frame)
  (list
   'LDL 5                               ; 5 = <rp>
   'LDNLP -2
   'STL 5                               ; 5 = <rp>
   'LDL 5                               ; 5 = <rp>
   'LDNL 1
   'STL 3                               ; 3 = <base>
   'LDL 5                               ; 5 = <rp>
   'LDNL 0
   'OPR 6                               ; GCALL
   ))
```

Issues:

Entry-code: push the return address and the old frame pointer value on the return stack. Check for return stack overflow. Exit-code: pop old frame pointer value and return address from return stack and jump to return address.

Comments:

memtop points two cells below valid memory, thus checking for return stack overflow after saving the return address and the frame pointer value is valid.

Related Documents: Translation of procedure definitions by `CCdef` on page 79.

Name: Translation of statements

Part of: C^{int} to TASM–compiler

Specification:

This function is specified by a collection of rules each of which separately defines the compilation of a particular C^{int}–statement. In the following we will compare each of the rules separately with the code occuring in the single cases of the conditional refining the relation $\mathcal{CC}$ below. The construction of this function is described in detail in [Hof98a].

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond

     ((eql stmt '_ABORT) ...)

     ((consp stmt)
      (let ((op (car stmt)))
        (cond
         ((eql op 'PROGN) ...)
         ((eql op '_SETHEAP) ... )
         ((eql op '_SETSTACK) ... )
         ((eql op '_SETLOCAL) ... )
         ((eql op 'IF) ... )
         ((eql op 'DO) ... )
         ((eql op '_ALLOCATE) ... )
         ((eql op '_WRITE–CHAR) ... )
         ((eql op '_READ–CHAR) ... )
         ((eql op '_PEEK–CHAR) ... )
         (T                             ; proc call
          ... ))))

     (T (errorstop 26)))))
```

Name: Translation of IF–statements

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{stmt}}[\![\texttt{(IF } e\ s_1\ s_2\texttt{)}]\!]\,\varphi\,\sigma \ \supseteq\ \mathcal{CC}_{\text{expr}}[\![e]\!]\,\varphi\,\sigma \ \texttt{cj}\ |cs_1| + |\texttt{j}\ |cs_2||\ \texttt{(}cs_1\texttt{)}\ \texttt{j}\ |cs_2|\ \texttt{(}cs_2\texttt{)}$$

where $\mathcal{CC}_{\text{stmt}}[\![s_1]\!]\,\varphi\,\sigma \supseteq cs_1$, $\mathcal{CC}_{\text{stmt}}[\![s_2]\!]\,\varphi\,\sigma \supseteq cs_2$,
$e \in \texttt{<expression>}_{C^{\text{int}}}$, $s_1,\ s_2 \in \texttt{<statement>}_{C^{\text{int}}}$, and
$|cs_2|,\ |cs_1| + |\texttt{j}\ |cs_2|| \in \texttt{<word>}$.

$\mathcal{CC}.6$

$$\mathcal{CC}_{\text{stmt}}[\![\texttt{(IF } e\ s\texttt{)}]\!]\,\varphi\,\sigma \ \supseteq\ \mathcal{CC}_{\text{expr}}[\![e]\!]\,\varphi\,\sigma \ \texttt{cj}\ |cs|\ \texttt{(}cs\texttt{)}$$

where $\mathcal{CC}_{\text{stmt}}[\![s]\!]\,\varphi\,\sigma \supseteq cs$,
$e \in \texttt{<expression>}_{C^{\text{int}}}$, $s \in \texttt{<statement>}_{C^{\text{int}}}$, and
$|cs| \in \texttt{<word>}$.

$\mathcal{CC}.7$

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond
     ...
     ((consp stmt)
      (let ((op (car stmt)))
        (cond
          ...
          ((eql op 'IF)
           (let* ((pred (cadr stmt))
```

```
                (then (caddr stmt))
                (else (cadddr stmt))
                (compiled–then (CCstmt then genv framesize)))
            (if else
                (let* ((compiled–else (CCstmt else genv framesize)))
                  (append
                   (CCexpr pred genv framesize reglist)
                   (CG–ifthenelse reg0 compiled–then compiled–else)))
                (append
                 (CCexpr pred genv framesize reglist)
                 (CG–ifthen reg0 compiled–then)))))
          ...
          )))
     ...
     )))
```

Source–Code:

```
(defun CG–ifthenelse (>val compiled–then compiled–else)
  (let ((then–len (TC–instruction–length compiled–then))
        (else–len (TC–instruction–length compiled–else)))
    (list
     'CJ (+ then–len (TC–command–length else–len))
     compiled–then
     'J else–len
     compiled–else)))
```

Source–Code:

```
(defun CG–ifthen (>val compiled–then)
  (let ((then–len (TC–instruction–length compiled–then)))
    (list
     'CJ then–len
     compiled–then)))
```

Comments:

Jump distances are calculated in bytes of resulting machine code.

Related Documents: The definition of `TC-instruction-length` on page 120

Name: Translation of PROGN–statements

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{stmt}}[\![\texttt{(PROGN } s_1 \ \ldots \ s_n\texttt{)}]\!]\,\varphi\,\sigma \ \supseteq \ \mathcal{CC}_{\text{stmt}}[\![s_1]\!]\,\varphi\,\sigma \ \ldots \ \mathcal{CC}_{\text{stmt}}[\![s_n]\!]\,\varphi\,\sigma$$

where $s_1, \ \ldots \ s_n \in$ <statement>$_{C^{int}}$.

$\mathcal{CC}.8$

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond
      ...
      ((consp stmt)
       (let ((op (car stmt)))
         (cond
           ((eql op 'PROGN)   (CCstmts (cdr stmt) genv framesize))
           ...
           )))
      ...)))
```

Source–Code:

```
(defun CCstmts (stmts genv framesize)
  (let ((result NIL))
    (do ()
     ((null stmts))
     (setq result (rappend (CCstmt (car stmts) genv framesize) result))
     (setq stmts (cdr stmts)))
    (reverse result)))
```

Comments:

The function CCstmts implements the linear recursion on lists of C^{int}–statements.

Name: Translation of DO–statements

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{stmt}}[\![\,(\texttt{DO}\ s_1\ e\ s_2)\,]\!]\,\varphi\,\sigma \ \supseteq\ (cs_1\ ce)\ \texttt{eqc}\ 0\ \texttt{cj}\ l_1\ (cs_2)\ \texttt{j}\ -l_2$$

where

$$\begin{aligned}
\mathcal{CC}_{\text{stmt}}[\![\,s_1\,]\!]\,\varphi\,\sigma &\supseteq cs_1\\
\mathcal{CC}_{\text{stmt}}[\![\,s_2\,]\!]\,\varphi\,\sigma &\supseteq cs_2\\
\mathcal{CC}_{\text{expr}}[\![\,e\,]\!]\,\varphi\,\sigma &\supseteq ce\\
l_1 &= |cs_2\ \ \texttt{j}\ -l_2|\\
l_2 &= |cs_1\ ce\ \ \texttt{eqc}\ 0\ \texttt{cj}\ l_1\ \ cs_2\ \texttt{j}\ \ -l_2|
\end{aligned}$$

and $s_1,\ s_2 \in$ <statement>$_{C^{int}}$, $l_1, -l_2 \in$ <word>. $\mathcal{CC}.9$

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG-all-registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond
     ...
     ((consp stmt)
      (let ((op (car stmt)))
        (cond
          ...
          ((eql op 'DO)
           (let* ((cs1 (CCstmt (cadr stmt) genv framesize))
                  (ce  (CCexpr (caddr stmt) genv framesize reglist))
                  (cs2 (CCstmt (cadddr stmt) genv framesize)))
             (CG-do cs1 ce cs2)))
          ...
          )))
     ...
     )))
```

Source–Code:

```
(defun CG–do (cs1 ce cs2)
  (let* ((lcs1 (TC–instruction–length cs1))
         (lcs2 (TC–instruction–length cs2))
         (lce   (TC–instruction–length ce))
         (l1 0)
         (l2 0)
         (l1a 0)
         (l2a 0))

    (setq l1a (+ lcs2 (TC–command–length (– 0 l2))))
    (setq l2a (+ lcs1
                 (+ lce
                 (+ 1
                 (+ (TC–command–length l1)
                 (+ lcs2
                    (TC–command–length (– 0 l2)))))))))
    (setq l1 l1a)
    (setq l2 l2a)

    (setq l1a (+ lcs2 (TC–command–length (– 0 l2))))
    (setq l2a (+ lcs1
                 (+ lce
                 (+ 1
                 (+ (TC–command–length l1)
                 (+ lcs2
                   (TC–command–length (– 0 l2)))))))))

    (setq l1 l1a)
    (setq l2 l2a)

    (if (or
         (null
          (eql l1 (+ lcs2 (TC–command–length (– 0 l2)))))
         (null
          (eql l2
               (+ lcs1
               (+ lce
               (+ 1
               (+ (TC–command–length l1)
               (+ lcs2
                  (TC–command–length (– 0 l2))))))))))
        (errorstop 53))

    (list (append cs1 ce) 'EQC 0 'CJ l1 cs2 'J (– 0 l2))))
```

Comments:

The calculation of l_1 and l_2 follows the way described in chapter 3 of [Hof98a].

Name: Translation of _SETHEAP–statements

Part of: C^{int} to TASM–compiler

Specification:

$\mathcal{CC}_{\text{stmt}}[\![$ (_SETHEAP e_1 e_2) $]\!]\varphi\sigma$	$\supseteq$	$\mathcal{CC}_{\text{expr}}[\![e_1]\!]\varphi\sigma$; value is in register A $\mathcal{CC}_{\text{expr}}[\![e_2]\!]\varphi\sigma$; value is now in register B ; and index in register A ldl *heaptop* ; check that index is csub0 ; within heap bounds ldl *heap* ; get heap start address wsub ; calculate memory address stnl 0 ; store value

where $e_1, e_2 \in$ <expression>$_{C^{int}}$.

$\mathcal{CC}$.10

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond
      ...

      ((consp stmt)
       (let ((op (car stmt)))
         (cond
           ...
           ((eql op '_SETHEAP)
            (let ((e0 (cadr stmt))
                  (e1 (caddr stmt)))
              (append
               (CCexpr e0 genv framesize reglist)
               (append
                (CCexpr e1 genv framesize restregs)
                (CG–setheap reg0 reg1)))))
           ...
           )))
      ...
      )))
```

Source–Code:

```
(defun CG-setheap (>val >offset)
  (list
   'LDL 2                                ; 2 = <heaptop>
   'OPR 19                               ; CSUB0
   'LDL 1                                ; 1 = <heap>
   'OPR 10                               ; WSUB
   'STNL 0
   ))
```

Name: Translation of _SETLOCAL–statements

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{stmt}}[\![\,(\texttt{\_SETLOCAL}\ e\ i)\,]\!]\,\varphi\,\sigma \quad \supseteq \quad \begin{array}{ll} \mathcal{CC}_{\text{expr}}[\![\,e\,]\!]\,\varphi\,\sigma & \text{; value is in register A} \\ \texttt{ldl}\ \textit{base} & \text{; load frame pointer} \\ \texttt{stnl}\ \text{i} & \text{; store value to} \end{array}$$

where $e \in \langle\texttt{expression}\rangle_{C^{int}}$ and $i \in \langle\texttt{word}\rangle$, $0 \le i < \sigma$.

$\mathcal{CC}.11$

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond
      ...

      ((consp stmt)
       (let ((op (car stmt)))
         (cond
           ...
           ((eql op '_SETLOCAL)
            (let ((e0 (cadr stmt))
                  (e1 (caddr stmt)))
              (if (>= e1 framesize)
                  (errorstop 25))
              (append
               (CCexpr e0 genv framesize reglist)
               (CG–setlocal reg0 e1))))
           ...
           )))
      ...
      )))
```

Source–Code:

```
(defun CG–setlocal (>val offset)
  (list
   'LDL 3                               ; 3 = <base>
   'STNL offset
   ))
```

Name: Translation of _SETSTACK–statements

Part of: C^{int} to TASM–compiler

Specification:

$\mathcal{CC}_{\text{stmt}}[\![$ (_SETSTACK e_1 e_2) $]\!]\,\varphi\,\sigma$ $\supseteq$

```
CCexpr[[e1]] φ σ   ; value is in register A
CCexpr[[e2]] φ σ   ; value is in register B
                   ; index is in register A
xdbl               ; check that index is
rev                ; non–negative
cj 2
seterr
add                ; calculate address
ldlp stack
wsub
stl temp
ldl base           ; check base
ldl temp
gt
cj -11             ; jump to seterr above
ldl temp           ; store value in location
stnl 0
```

where $e_1, e_2 \in$ <expression>$_{C^{int}}$.

$\mathcal{CC}$.12

$\mathcal{CC}_{\text{stmt}}[\![$ (_SETSTACK e i) $]\!]\,\varphi\,\sigma$ $\supseteq$

```
CCexpr[[e]] φ σ   ; value is in register A
stl stack+i       ; store value
```

where $e \in$ <expression>$_{C^{int}}$, $i \in$ <word>, $\varphi =$ <ψ, *s-size*, *h-size*>, and $0 \leq i <$ *s-size*.

$\mathcal{CC}$.13

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))          ; can be NIL, if too few registers
         (reg1 (car restregs)))        ; can be NIL, if too few registers
    (cond
```

```
...

((consp stmt)
 (let ((op (car stmt)))
   (cond
     ...
     ((eql op '_SETSTACK)
      (let ((e0 (cadr stmt))
            (e1 (caddr stmt)))
        (append
         (CCexpr e0 genv framesize reglist)
         (if (integerp e1)
             (let ((s-size (cadr genv)))
               (if (>= e1 s-size)
                   (errorstop 55)
                   (CG-setstacki reg0 e1)))
             (append
              (CCexpr e1 genv framesize restregs)
              (CG-setstack reg0 reg1))))))
     ...
     )))
...
)))
```

Source–Code:

```
(defun CG-setstacki (>val offset)
  (list
   'STL (+ offset 13)                    ; 13 = <stack>
   ))
```

Source–Code:

```
(defun CG-setstack (>val >offset)
  (list
   'OPR 29                               ; XDBLE
   'OPR 0                                ; REV
   'CJ 2
   'OPR 16                               ; SETERR
   'OPR 5                                ; ADD (CJ doesn't discard FALSE)
   'LDLP 13                              ; 13 = <stack>
   'OPR 10                               ; WSUB
   'STL 11                               ; 11 = <temp>
   'LDL 3                                ; 3 = <base>
   'LDL 11                               ; 11 = <temp>
   'OPR 9                                ; GT
   'CJ -11
   'LDL 11                               ; 11 = <temp>
   'STNL 0
   ))
```

Name: Translation of procedure calls

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{stmt}}[\![(f\ \ i)]\!]\varphi\sigma \ \supseteq\ \begin{array}{l}\texttt{ldc}\ i\\ \texttt{ldl}\ \textit{start}\\ \texttt{ldnl}\ \psi(f)\\ \texttt{gcall}\end{array}$$

where $f \in$ <ident>, $\varphi = <\psi, \textit{s-size}, \textit{h-size}>$, and $i, \psi(f) \in$ <word>.

$\mathcal{CC}$.14

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond
     ...
     ((consp stmt)
      (let ((op (car stmt)))
        (cond
          ...
          (T                            ; proc call
           (CG–call op (cadr stmt) genv)))))
     ...
     )))
```

Source–Code:

```
(defun CG–call (name offset genv)
  (list
   'LDC offset
   'LDL 0                               ; 0 = <start>
   'LDNL (TC–function–index name (car genv))
   'OPR 6                               ; GCALL
   ))
```

Comments:

The table of procedure start addresses is constructed by the boot loader.

Related Documents: The definition of `TC-function-index` on page 80.

Name: Translation of _ALLOCATE–statements

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{stmt}}[\![\texttt{(\_ALLOCATE } e\texttt{)}]\!]\,\varphi\,\sigma \;\supseteq\; \mathcal{CC}_{\text{expr}}[\![e]\!]\,\varphi\,\sigma$$

```
ldl heaptop add stl heaptop
ldl heaptop ldl heap wsub
ldl rstack gt
cj 2 seterr
ldl heaptop ldl quotetop gt
cj 2 seterr
```

where $e \in \langle\texttt{expression}\rangle_{C^{int}}$.

$\mathcal{CC}.15$

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond

     ...

     ((consp stmt)
      (let ((op (car stmt)))
        (cond
          ...
          ((eql op '_ALLOCATE)
           (append
            (CCexpr (cadr stmt) genv framesize reglist)
            (CG–allocate reg0)))
          ...
          )))
     ...
     )))
```

Source–Code:

```
(defun CG–allocate (>val)
  (list
   'LDL 2                              ; 2 = <heaptop>
   'OPR 5                              ; ADD
   'STL 2                              ; 2 = <heaptop>
   'LDL 2                              ; 2 = <heaptop>
   'LDL 1                              ; 1 = <heap>
   'OPR 10                             ; WSUB
   'LDL 6                              ; 6 = <rstack>
   'OPR 9                              ; GT
   'CJ 2
   'OPR 16                             ; SETERR
   'LDL 10                             ; 10 = <quotetop>
   'LDL 2                              ; 2 = <heaptop>
   'OPR 9                              ; GT
   'CJ 2
   'OPR 16                             ; SETERR
   ))
```

Name: Translation of _READ-CHAR–statements

Part of: C^{int} to TASM–compiler

Specification:

$\mathcal{CC}_{\text{expr}}[\![\,(\texttt{\_READ-CHAR}\ i)\,]\!]\,\varphi\,\sigma \supseteq$

```
ldc lastchar eqc -1 cj 6
ldc 0 stl lastchar ldlp lastchar
ldl inchan ldc 1 in
ldl lastchar ldl base stnl i
ldc -1 stl lastchar
```

where $i \in$ <word> and $0 \leq i < \sigma$.

$\mathcal{CC}$.16

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond

     ...

     ((consp stmt)
      (let ((op (car stmt)))
        (cond
          ...
          ((eql op '_READ–CHAR) (CG–read–char (cadr stmt)))
          ...
        )))
     ...
     )))
```

Source–Code:

```
(defun CG–read–char (offset)
  (append
   (CG–peek–char offset)
   (list
    'LDC –1
    'STL 12                             ; 12 = <lastch>
    )))
```

Comments:

The system variable *lastch* is used to buffer characters fetched from the input medium which must be delivered (possibly several times) to the program. The generated code first performs _PEEK-CHARACTER to get the next character and delivers it to the program. *lastch* is set to -1 to signal that no character is buffered.

Related Documents: The translation of _PEEK-CHAR on the next page.

Name: Translation of _PEEK-CHAR–statements

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{expr}}[\![\,\texttt{(\_PEEK-CHAR}\ i\texttt{)}\,]\!]\,\varphi\,\sigma \ \supseteq$$

```
ldc lastchar eqc -1 cj 6
ldc 0 stl lastchar ldlp lastchar
ldl inchan ldc 1 in
ldl lastchar ldl base stnl i
```

where $i \in$ <word> and $0 \leq i < \sigma$.

$\mathcal{CC}.17$

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond

     ...

     ((consp stmt)
      (let ((op (car stmt)))
        (cond
          ...
          ((eql op '_PEEK–CHAR) (CG–peek–char (cadr stmt)))
          ...
        )))
     ...
     )))
```

Source–Code:

```
(defun CG–peek–char (offset)
  (list
   'LDL 12                              ; 12 = <lastch>
   'EQC –1
   'CJ 6
   'LDC 0
   'STL 12                              ; 12 = <lastch>
   'LDLP 12                             ; 12 = <lastch>
   'LDL 9                               ; 9 = <inchan>
   'LDC 1
   'OPR 7
   'LDL 12                              ; 12 = <lastch>
```

```
'LDL 3                                  ; 3 = <base>
'STNL offset
))
```

Comments:

If no character is buffered (in *lastch*), then the next character is fetched from the input medium. If a character is buffered it is simply returned.

Related Documents: The translation of _READ-CHAR on the page 97.

Name: Translation of _WRITE-CHAR–statements

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{stmt}}[\![\texttt{(\_WRITE-CHAR } i\texttt{)}]\!]\,\varphi\,\sigma \;\supseteq\; \begin{array}{l}\texttt{ldl } \textit{base}\ \texttt{ldnlp } i\\ \texttt{ldl } \textit{outchan}\ \texttt{ldc 1}\\ \texttt{out}\end{array}$$

where $i \in$ <word> and $0 \leq i < \sigma$.

$\mathcal{CC}$.18

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG-all-registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond

     ...

     ((consp stmt)
      (let ((op (car stmt)))
        (cond
          ...
          ((eql op '_WRITE-CHAR) (CG-write-char (cadr stmt)))
          ...
        )))
     ...
     )))
```

Source–Code:

```
(defun CG-write-char (offset)
  (list
   'LDL 3                               ; 3 = <base>
   'LDNLP offset
   'LDL 8                               ; 8 = <outchan>
   'LDC 1
   'OPR 11                              ; OUT
   ))
```

Name: Translation of _ABORT–statements

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{stmt}}[\![\texttt{\_ABORT}]\!]\varphi\sigma \;\supseteq\; \texttt{seterr}$$

$\mathcal{CC}.19$

Source–Code:

```
(defun CCstmt (stmt genv framesize)
  (let* ((reglist (CG–all–registers))
         (restregs (cdr reglist))
         (reg0 (car reglist))           ; can be NIL, if too few registers
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond

     ((eql stmt '_ABORT) (CG–abort))

     ...
     )))
```

Source–Code:

```
(defun CG–abort ()
  (list
   'OPR 16                              ; SETERR
   ))
```

Name: Translation of expressions

Part of: C^{int} to TASM–compiler

Specification:

This function is specified by a collection of rules each of which separately defines the compilation of a particular C^{int}–expression. In the following we will compare each of the rules separately with the code occuring in the single cases of the conditional refining the relation $\mathcal{CC}$ below. The construction of this function is described in detail in [Hof98a].

Source–Code:

```
(defun CCexpr (expr genv framesize reglist)

  (if (null reglist)
      (errorstop 27))

  (let* ((restregs (cdr reglist))
         (reg0 (car reglist))           ; is always valid
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond

      ((eql expr '_STACKTOP) ... )
      ((eql expr '_HEAPTOP) ... )
      ((eql expr '_QUOTETOP) ... )

      ((consp expr)
       (let ((op (car expr))
             (e0 (cadr expr))           ; may not be present -> NIL
             (e1 (caddr expr)))         ; may not be present -> NIL
         (cond

           ((eql op '_UNAVAILABLE) ... )
           ((eql op '_2*) ... )
           ((eql op '_*) ... )
           ((eql op '_+) ... )
           ((eql op '_-) ... )
           ((eql op '_div) ... )
           ((eql op '_rem) ... )
           ((eql op '_<) ... )
           ((eql op '_>=) ... )
           ((eql op '_=) ... )
           ((eql op '_!=) ... )
           ((eql op '_HEAP) ... )
           ((eql op '_STACK) ... )
           ((eql op '_LOCAL) ... )
           (T (errorstop 29)))))

      ((integerp expr)  ... )

      (T (errorstop 30)))))
```

Name: Translation of arithmetic expressions

Part of: C^{int} to TASM–compiler

Specification:

$$
\begin{array}{rcl}
\mathcal{CC}_{\text{expr}}[\![\texttt{(\_2* } e \texttt{)}]\!]\varphi\sigma & \supseteq & \mathcal{CC}_{\text{expr}}[\![e]\!]\varphi\sigma \texttt{ ldc 2 mul} \\
\mathcal{CC}_{\text{expr}}[\![\texttt{(\_* } e\ s \texttt{)}]\!]\varphi\sigma & \supseteq & \mathcal{CC}_{\text{expr}}[\![e]\!]\varphi\sigma\ \mathcal{CC}_{\text{expr}}[\![s]\!]\varphi\sigma \texttt{ mul} \\
\mathcal{CC}_{\text{expr}}[\![\texttt{(\_+ } e\ s \texttt{)}]\!]\varphi\sigma & \supseteq & \mathcal{CC}_{\text{expr}}[\![e]\!]\varphi\sigma\ \mathcal{CC}_{\text{expr}}[\![s]\!]\varphi\sigma \texttt{ add} \\
\mathcal{CC}_{\text{expr}}[\![\texttt{(\_- } e\ s \texttt{)}]\!]\varphi\sigma & \supseteq & \mathcal{CC}_{\text{expr}}[\![e]\!]\varphi\sigma\ \mathcal{CC}_{\text{expr}}[\![s]\!]\varphi\sigma \texttt{ sub} \\
\mathcal{CC}_{\text{expr}}[\![\texttt{(\_DIV } e\ s \texttt{)}]\!]\varphi\sigma & \supseteq & \mathcal{CC}_{\text{expr}}[\![e]\!]\varphi\sigma\ \mathcal{CC}_{\text{expr}}[\![s]\!]\varphi\sigma \texttt{ div} \\
\mathcal{CC}_{\text{expr}}[\![\texttt{(\_REM } e\ s \texttt{)}]\!]\varphi\sigma & \supseteq & \mathcal{CC}_{\text{expr}}[\![e]\!]\varphi\sigma\ \mathcal{CC}_{\text{expr}}[\![s]\!]\varphi\sigma \texttt{ rem} \\
\mathcal{CC}_{\text{expr}}[\![\texttt{(\_< } e\ s \texttt{)}]\!]\varphi\sigma & \supseteq & \mathcal{CC}_{\text{expr}}[\![e]\!]\varphi\sigma\ \mathcal{CC}_{\text{expr}}[\![s]\!]\varphi\sigma \texttt{ rev gt} \\
\mathcal{CC}_{\text{expr}}[\![\texttt{(\_>= } e\ s \texttt{)}]\!]\varphi\sigma & \supseteq & \mathcal{CC}_{\text{expr}}[\![e]\!]\varphi\sigma\ \mathcal{CC}_{\text{expr}}[\![s]\!]\varphi\sigma \texttt{ rev gt eqc 0} \\
\mathcal{CC}_{\text{expr}}[\![\texttt{(\_!= } e\ s \texttt{)}]\!]\varphi\sigma & \supseteq & \mathcal{CC}_{\text{expr}}[\![e]\!]\varphi\sigma\ \mathcal{CC}_{\text{expr}}[\![s]\!]\varphi\sigma \texttt{ diff} \\
\mathcal{CC}_{\text{expr}}[\![\texttt{(\_= } e\ s \texttt{)}]\!]\varphi\sigma & \supseteq & \mathcal{CC}_{\text{expr}}[\![e]\!]\varphi\sigma\ \mathcal{CC}_{\text{expr}}[\![s]\!]\varphi\sigma \texttt{ diff eqc 0}
\end{array}
$$

where $e, s \in \texttt{<expression>}_{C^{int}}$.

$\mathcal{CC}.20$

Source–Code:

```
(defun CCexpr (expr genv framesize reglist)
  (if (null reglist)
      (errorstop 27))
  (let* ((restregs (cdr reglist))
         (reg0 (car reglist))           ; is always valid
         (reg1 (car restregs)))         ; can be NIL, if too few registers
    (cond
      ...
      ((consp expr)
       (let ((op (car expr))
             (e0 (cadr expr))           ; may not be present -> NIL
             (e1 (caddr expr)))         ; may not be present -> NIL
         (cond
           ...

           ((eql op '_2*)
            (append
             (CCexpr e0 genv framesize reglist)
             (CG-2* reg0)))

           ((eql op '_*)
            (append
             (CCexpr e0 genv framesize reglist)
             (append
```

```
    (CCexpr e1 genv framesize restregs)
    (CG-* reg0 reg1))))

((eql op '_+)
 (append
  (CCexpr e0 genv framesize reglist)
  (append
   (CCexpr e1 genv framesize restregs)
   (CG-+ reg0 reg1))))

((eql op '_-)
 (append
  (CCexpr e0 genv framesize reglist)
  (append
   (CCexpr e1 genv framesize restregs)
   (CG-- reg0 reg1))))

((eql op '_div)
 (append
  (CCexpr e0 genv framesize reglist)
  (append
   (CCexpr e1 genv framesize restregs)
   (CG-div reg0 reg1))))

((eql op '_rem)
 (append
  (CCexpr e0 genv framesize reglist)
  (append
   (CCexpr e1 genv framesize restregs)
   (CG-rem reg0 reg1))))

((eql op '_<)
 (append
  (CCexpr e0 genv framesize reglist)
  (append
   (CCexpr e1 genv framesize restregs)
   (CG-< reg0 reg1))))

((eql op '_>=)
 (append
  (CCexpr e0 genv framesize reglist)
  (append
   (CCexpr e1 genv framesize restregs)
   (CG->= reg0 reg1))))

((eql op '_=)
 (append
  (CCexpr e0 genv framesize reglist)
  (append
   (CCexpr e1 genv framesize restregs)
   (CG-= reg0 reg1))))

((eql op '_!=)
```

```
           (append
            (CCexpr e0 genv framesize reglist)
            (append
             (CCexpr e1 genv framesize restregs)
             (CG-!= reg0 reg1))))
          ...

          (T (errorstop 29)))))
     ...
     )))
```

Source–Code:

```
(defun CG-2* (>val>)
  (list
   'LDC 2
   'OPR 83                              ; MUL
   ))
```

Source–Code:

```
(defun CG-* (>val1> >val2)
  (list
   'OPR 83                              ; MUL
   ))
```

Source–Code:

```
(defun CG-+ (>val1> >val2)
  (list
   'OPR 5                               ; ADD
   ))
```

Source–Code:

```
(defun CG-- (>val1> >val2)
  (list
   'OPR 12                              ; SUB
   ))
```

Source–Code:

```
(defun CG-div (>val1> >val2)
  (list
   'OPR 44                              ; DIV
   ))
```

Source–Code:

```
(defun CG–rem (>val1> >val2)
  (list
   'OPR 31                              ; REM
   ))
```

Source–Code:

```
(defun CG–< (>val1> >val2)
  (list
   'OPR 0                               ; REV
   'OPR 9                               ; GT
   ))
```

Source–Code:

```
(defun CG–>= (>val1> >val2)
  (list
   'OPR 0                               ; REV
   'OPR 9                               ; GT
   'EQC 0
   ))
```

Source–Code:

```
(defun CG–!= (>val1> >val2)
  (list
   'OPR 4                               ; DIFF
   ))
```

Source–Code:

```
(defun CG–= (>val1> >val2)
  (list
   'OPR 4                               ; DIFF
   'EQC 0
   ))
```

Comments:

The transputer arithmetic instruction do overflow checking on the fly so no explicit code to detect ressource violations is necessary. For this to work, the transputer *haltonerror*-bit must be set during program execution. This is done by the bootloader (cf. [GH98b]). *haltonerror* is never modified after that.

Name: Translation of constants

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{expr}}[\![i]\!]\,\varphi\,\sigma \quad \supseteq \quad \texttt{ldc}\ i$$

where $i \in$ <word>.

$\mathcal{CC}.21$

Source–Code:

```
(defun CCexpr (expr genv framesize reglist)

  (if (null reglist)
      (errorstop 27))

  (let* ((restregs (cdr reglist))
         (reg0 (car reglist))           ; is always valid
         (reg1 (car restregs)))         ; can be NIL, if too few registers

    (cond
      ...
      ((integerp expr)  (CG-int reg0 expr))
      ...
      )))
```

Source–Code:

```
(defun CG-int (val2> val1)
  (list
   'LDC val1
   ))
```

Comments:

The function `TC-check-code` is called with the TASM-code generated for the entire function definition to assure that all constants used are in the approporiate range of numbers representable in TASM.

Related Documents: The translation of function definitions on page 79.

Name: Translation of _LOCAL–expressions

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{expr}}[\![\,\texttt{(\_LOCAL}\ i\texttt{)}\,]\!]\,\varphi\,\sigma \quad \supseteq \quad \texttt{ldl}\ \ \textit{base}\ \ \ \texttt{ldnl}\ i$$

where $i \in$ <word> and $0 \leq i < \sigma$.

$\mathcal{CC}.22$

Source–Code:

```
(defun CCexpr (expr genv framesize reglist)

  (if (null reglist)
      (errorstop 27))

  (let* ((restregs (cdr reglist))
         (reg0 (car reglist))           ; is always valid
         (reg1 (car restregs)))         ; can be NIL, if too few registers

    (cond
      ...
      ((consp expr)
       (let ((op (car expr))
             (e0 (cadr expr))            ; may not be present -> NIL
             (e1 (caddr expr)))          ; may not be present -> NIL
         (cond
           ...
           ((eql op '_LOCAL)
            (if (>= e0 framesize)
                (errorstop 28))
            (CG-local reg0 e0))
           ...
           )))
      ...
      )))
```

Source–Code:

```
(defun CG-local (val> offset)
  (list
   'LDL 3                               ; 3 = <base>
   'LDNL offset
   ))
```

Name: Translation of _STACKTOP–expressions

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{expr}}[\![\,\texttt{\_STACKTOP}\,]\!]\,\varphi\,\sigma \quad \supseteq$$

```
stl temp
ldl base
ldlp stack
diff
wcnt
rev
stl temp2
ldl temp
rev
```

$\mathcal{CC}.23$

Source–Code:

```
(defun CCexpr (expr genv framesize reglist)

  (if (null reglist)
      (errorstop 27))

  (let* ((restregs (cdr reglist))
         (reg0 (car reglist))           ; is always valid
         (reg1 (car restregs)))         ; can be NIL, if too few registers

    (cond

     ((eql expr '_STACKTOP) (CG-stacktop reg0))

     ...
     )))
```

Source–Code:

```
(defun CG–stacktop (val>)
  (list
   'STL 11                              ; 11 = <temp> (save Areg)
   'LDL 3                               ; 3 = <base>
   'LDLP 13                             ; 13 = <stack>
   'OPR 4                               ; DIFF
   'OPR 63                              ; WCNT
   'OPR 0                               ; REV
   'STL 7                               ; 7 = <temp2> (discard byteselector)
   'LDL 11                              ; 11 = <temp> (restore Areg)
   'OPR 0                               ; REV
   ))
```

Comments:

The generated code preserves the values which are strored in the transputer registers Areg and Breg before execution.

Name: Translation of _UNAVAILABLE–expressions

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{expr}}[\![\,(\texttt{\_UNAVAILABLE}\ e)\,]\!]\,\varphi\,\sigma \;\supseteq\; \mathcal{CC}_{\text{expr}}[\![\,e\,]\!]\,\varphi\,\sigma$$
ldl *heaptop*
add
ldl *heap*
wsub
ldl *rstack*
gt

where $e \in \langle\texttt{expression}\rangle_{C^{\text{int}}}$.

$\mathcal{CC}.24$

Source–Code:

```
(defun CG-unavailable (>val>)
  (list
   'LDL 2                              ; 2 = <heaptop>
   'OPR 5                              ; ADD
   'LDL 1                              ; 1 = <heap>
   'OPR 10                             ; WSUB
   'LDL 6                              ; 6 = <rstack>
   'OPR 9                              ; GT
   ))
```

Name: Translation of _HEAP–expressions

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{expr}}[\![\,(\texttt{\_HEAP } e)\,]\!]\,\varphi\,\sigma \;\supseteq\; \mathcal{CC}_{\text{expr}}[\![\,e\,]\!]\,\varphi\,\sigma$$
ldl *heaptop*
csub0
ldl *heap*
wsub
ldnl 0

where $e \in$ <expression>$_{C^{int}}$.

$\mathcal{CC}.25$

Source–Code:

```
(defun CCexpr (expr genv framesize reglist)

  (if (null reglist)
      (errorstop 27))

  (let* ((restregs (cdr reglist))
         (reg0 (car reglist))           ; is always valid
         (reg1 (car restregs)))         ; can be NIL, if too few registers

    (cond
      ...
      ((consp expr)
       (let ((op (car expr))
             (e0 (cadr expr))            ; may not be present -> NIL
             (e1 (caddr expr)))          ; may not be present -> NIL
         (cond
           ...
           ((eql op '_HEAP)
            (append
             (CCexpr e0 genv framesize reglist)
             (CG-heap reg0)))
           ...
           )))
      ...
      )))
```

Source–Code:

```
(defun CG-heap (>val>)
  (list
   'LDL 2                                ; 2 = <heaptop>
   'OPR 19                               ; CSUB0
   'LDL 1                                ; 1 = <heap>
   'OPR 10                               ; WSUB
   'LDNL 0
   ))
```

Name: Translation of _STACK–expression

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{expr}}[\![\texttt{(\_STACK } e\texttt{)}]\!]\,\varphi\,\sigma \;\supseteq\; \mathcal{CC}_{\text{expr}}[\![e]\!]\,\varphi\,\sigma$$
`xdbl rev cj 2 seterr`
`add`
`ldlp` *stack* `wsub stl` *temp*
`ldl` *base* `ldl` *temp* `gt cj -11`
`ldl` *temp*
`ldnl 0`

where $e \in$ <expression>$_{C^{int}}$.

$\mathcal{CC}$.26

$$\mathcal{CC}_{\text{expr}}[\![\texttt{(\_STACK i)}]\!]\,\varphi\,\sigma \;\supseteq\; \texttt{ldl } \textit{stack}{+}i$$

where $i \in$ <word>, $\varphi = <\psi$, *s-size*, *h-size*> and $0 \leq i <$ *s-size*.

$\mathcal{CC}$.27

Source–Code:

```
(defun CCexpr (expr genv framesize reglist)

  (if (null reglist)
      (errorstop 27))

  (let* ((restregs (cdr reglist))
         (reg0 (car reglist))           ; is always valid
         (reg1 (car restregs)))         ; can be NIL, if too few registers

    (cond
      ...
      ((consp expr)
       (let ((op (car expr))
             (e0 (cadr expr))           ; may not be present -> NIL
             (e1 (caddr expr)))         ; may not be present -> NIL
         (cond
           ...
```

```
          ((eql op '_STACK)
           (if (integerp e0)
               (let ((s-size (cadr genv)))
                 (if (>= e0 s-size)
                     (errorstop 56)
                     (CG-stacki reg0 e0)))
               (append
                (CCexpr e0 genv framesize reglist)
                (CG-stack reg0))))
          ...
          )))
     ...
     )))
```

Source–Code:

```
(defun CG-stack (>val>)
  (list
   'OPR 29                              ; XDBLE
   'OPR 0                               ; REV
   'CJ 2
   'OPR 16                              ; SETERR
   'OPR 5                               ; ADD (CJ doesn't discard FALSE)
   'LDLP 13                             ; 13 = <stack>
   'OPR 10                              ; WSUB
   'STL 11                              ; 11 = <temp>
   'LDL 3                               ; 3 = <base>
   'LDL 11                              ; 11 = <temp>
   'OPR 9                               ; GT
   'CJ -11
   'LDL 11                              ; 11 = <temp>
   'LDNL 0
   ))
```

Source–Code:

```
(defun CG-stacki (val> offset)
  (list
   'LDL (+ 13 offset)                   ; 13 = <stack>
   ))
```

Name: Translation of _HEAPTOP–expressions

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{expr}}[\![\,\texttt{(\_HEAPTOP)}\,]\!]\,\varphi\,\sigma \;\supseteq\; \texttt{ldl}\ \textit{heaptop}$$

$\mathcal{CC}.28$

Source–Code:

```
(defun CCexpr (expr genv framesize reglist)

  (if (null reglist)
      (errorstop 27))

  (let* ((restregs (cdr reglist))
         (reg0 (car reglist))           ; is always valid
         (reg1 (car restregs)))         ; can be NIL, if too few registers

    (cond
      ...
      ((eql expr '_HEAPTOP)  (CG-heaptop reg0))
      ...
      )))
```

Source–Code:

```
(defun CG-heaptop (val>)
  (list
   'LDL 2                               ; 2 = <heaptop>
   ))
```

Name: Translation of _QUOTETOP–expressions

Part of: C^{int} to TASM–compiler

Specification:

$$\mathcal{CC}_{\text{expr}}[\![\,\texttt{(\_QUOTETOP)}\,]\!]\,\varphi\,\sigma \;\supseteq\; \texttt{ldl}\ \textit{quotetop}$$

$\mathcal{CC}.29$

Source–Code:

```
(defun CCexpr (expr genv framesize reglist)

  (if (null reglist)
      (errorstop 27))

  (let* ((restregs (cdr reglist))
         (reg0 (car reglist))           ; is always valid
         (reg1 (car restregs)))         ; can be NIL, if too few registers

    (cond
      ...
      ((eql expr '_QUOTETOP) (CG-quotetop reg0))
      ...
      )))
```

Source–Code:

```
(defun CG-quotetop (val>)
  (list
   'LDL 10                              ; 10 = <quotetop>
   ))
```

Name: CG-all-registers

Part of: C^{int} to TASM–compiler

Specification:

No explicit specification

Source–Code:

```
(defun CG–all–registers ()
  (list 1 2 3))
```

Issues:

Return list of available registers for statement translation.

Comments:

In the transputer code generator only the number of available registers is important. for expression translation the available registers are passed in the formal parameter `reglist`. Each subexpression consumes one register and code generation takes place according to the stack principle suitable for the transputer with its expression mini stack of registers Areg, Breg and Creg. If the registers are exhausted then the compiler terminates irregularly.

Related Documents: The definition of `CCexpr` on page 103.

Name: TC-instruction-length

Part of: C^{int} to TASM–compiler

Specification:

The length (in bytes) of a TASM–instruction sequence is determined by the number of instructions and the length of the associated `pfix/nfix`–chain for each instruction. Based on *length* we define the length $|\cdot| : \texttt{<op>}_{\text{TASM}}{}^{*} \to \mathbb{N}_0$ of a sequence of TASM–instructions by:

$$\begin{aligned} |\varepsilon| &:= 0 \\ |\mathit{opr}\ \mathit{arg}| &:= \mathit{length}(\mathit{arg}) + 1 \\ |\mathit{opr}_1\ \mathit{arg}_1\ \ldots\ \mathit{opr}_n\ \mathit{arg}_n| &:= |\mathit{opr}_1\ \mathit{arg}_1| + \ldots + |\mathit{opr}_n\ \mathit{arg}_n| \end{aligned}$$

Actually, instruction sequences can be nested arbitrarily (cf. specification of $\mathcal{CA}_{\text{body}}$ on page 127). This specification does not mention parentheses.

Source–Code:

```
(defun TC–instruction–length (codelist)
  (let ((result 0))
    (do ()
        ((null codelist))
      (cond
        ((null (car codelist))
         (setq codelist (cdr codelist)))
        ((consp (car codelist))
         (setq result (+ result (TC–instruction–length (car codelist))))
         (setq codelist (cdr codelist)))
        (T
         (let ((operand (cadr codelist)))
           (setq result (+ result (TC–command–length operand)))
           (setq codelist (cddr codelist))))))
    result))
```

Issues:

Calculate the length of the instruction sequence `codelist` in bytes of TC–machine code.

Comments:

The implementation handles also nested sequences.

Specification:

$$length(e) := \begin{cases} \lfloor \log_{16}\ e \rfloor & \text{, if } e \geq 0 \\ 1 & \text{, if } -16 \leq e < 0 \\ \lfloor \log_{16}\ \overline{e} \rfloor & \text{, if } e < -16. \end{cases}$$

Source–Code:

```
(defun TC–command–length (operand)
  ;; length of necessary pfix/nfix chain.
  (cond
   ((< operand 0)
    (+ 1 (TC–command–length (floor (– –1 operand) 16))))
   ((< operand 16) 1)
   (T
    (+ 1 (TC–command–length (floor operand 16))))))
```

Issues:

Calculate the length of a TASM–instruction with operand `operand` in bytes of TC–machine code.

Related Documents: The translation of `IF`–statements on page 84 and of `DO`–statemements on page 87.

Chapter 6

TASM to TC

The compiling relation between TASM and TC is specified in [GH98b] to be a syntactical mapping between TASM source programs and TC target programs. It is defined recursively using several auxiliary relations which correspond to the TASM source program structure. The types are as follows:

$$
\begin{array}{rcl}
\mathcal{CA} & \in & \texttt{<program>}_{\text{TASM}} \rightharpoonup \texttt{<program>}_{\text{TC}} \\
\mathcal{CA}_{\text{def}} & \in & \texttt{<module>}_{\text{TASM}} \rightharpoonup \texttt{<module>}_{\text{TC}} \\
\mathcal{CA}_{\text{body}} & \in & \texttt{<body>} \rightharpoonup \texttt{<byte>}^* \\
\mathcal{CA}_{\text{op}} & \in & \texttt{<mnemonic>} \times \texttt{<arg>} \rightharpoonup \texttt{<byte>}^* \\
 & & \\
\mathit{prefix} & : & \texttt{<nibble>} \times \texttt{<arg>} \to \texttt{<byte>}^* \\
\mathit{assemble\_op} & : & \texttt{<mnemonic>} \to \texttt{<nibble>}
\end{array}
$$

Again, these relations are now compared to ComLisp implementations (cf. page 17).

6.1 Assembler: Code Review

Name: ca.definition

Part of: TASM to TC–compiler

Specification:

$$\mathcal{CA}[\![\,(l)\ m_1\ \cdots\ m_n\,]\!] \quad \supseteq \quad (\,\mathcal{CA}_{\mathrm{def}}[\![\,m_1\,]\!]\ \cdots\ \mathcal{CA}_{\mathrm{def}}[\![\,m_n\,]\!]\ l\,)$$

where $m_1, \ldots, m_n \in$ <module>$_{\mathrm{TASM}}$, $l \in$ <index>.

$\mathcal{CA}.1$

Source–Code:

```
(defun CA (p)
  (let* ((asmdecls (car p))
         (asmdefs (cdr p))
         (asmenv–bindecl–bindefs (CAdecl asmdecls))
         (asmenv (car asmenv–bindecl–bindefs)))
    (append (CAdefs asmdefs asmenv)
            (cadr asmenv–bindecl–bindefs))))
```

Source–Code:

```
(defun CAdecl (decls)
  (list NIL (list (car decls))))
```

Name: cadefs.definition

Part of: TASM to TC–compiler

Specification:

No explicit specification

Source–Code:

```
(defun CAdefs (defs asm–env)
  (if (null defs) NIL
      (cons (CAdef (car defs))
            (CAdefs (cdr defs) asm–env))))
```

Issues:

This function implements the implicit linear recursion, that appears in the specification of the transformation of sequences of C^{int} definitions

$$m_1 \ \ldots \ m_n \quad \text{to sequences} \quad \mathcal{CA}_{\text{def}}[\![\, m_1 \,]\!] \ \ldots \ \mathcal{CA}_{\text{def}}[\![\, m_n \,]\!] \,.$$

of compiled definitions. Sequences are implemented as lists.

Name: cadef.definition

Part of: TASM to TC–compiler

Specification:

$$\mathcal{CA}_{\text{def}}[\![\,(\texttt{\_DEFDATA}\ n\ i\ .\ w)\,]\!] \supseteq (\ i\ \texttt{W}\ |w|\ .\ w_{16}\ )$$
$$\mathcal{CA}_{\text{def}}[\![\,(\texttt{\_DEFCODE}\ n\ i\ .\ b)\,]\!] \supseteq (\ i\ \texttt{Z}\ |c|\ .\ c\ ), \quad \text{for } c = \mathcal{CA}_{\text{body}}[\![\,b\,]\!]$$

---

where $n \in$ <symbol>, $i \in$ <word>, $w \in$ <word>*, and $b \in$ <body>. w_{16} denotes the list of hexadecimal representations of the members of w. $(\ i\ \texttt{Z}\ |c|\ .\ c\ )$ denotes the list consisting of i, Z, $|c|$, followed by the elements of c, $(\ i\ \texttt{W}\ |w|\ .\ w_{16}\ )$ analogously.

$\mathcal{CA}.2$

Source–Code:

```
(defun CAdef (d)
  (let ((type (car d))
        (index (caddr d))
        (body (cdddr d)))
    (cond
      ((eql type '_DEFDATA)
       (list* index 'W (list-length body) body))
      ((eql type '_DEFCODE)
       (let ((bin-body (TC-tasm-body body)))
         (list* index 'Z (list-length bin-body) bin-body)))
      (T (errorstop 37)))))
```

Name: tc-tasm-body.definition

Part of: TASM to TC–compiler

Specification:

$$\mathcal{CA}_{\text{body}}[\![\ opr_1\ \ arg_1 \quad \cdots \quad (\cdots (\cdots) \cdots) \quad \cdots \quad (\cdots\ opr_n\ \ arg_n\ )\ ]\!] \supseteq \mathcal{CA}_{\text{op}}[\![\ opr_1, arg_1\ ]\!] \cdots \mathcal{CA}_{\text{op}}[\![\ opr_n, arg_n\ ]\!]$$

where $opr_1, \ldots, opr_n \in$ <mnemonic>, and $arg_1, \ldots, arg_n \in$ <word>.

$\mathcal{CA}.3$

Source–Code:

```
(defun TC–tasm–body (cmds)
  (let ((tc NIL))
    (do ()
        ((null cmds))
      (cond
        ((null (car cmds)) (setq cmds (cdr cmds)))
        ((consp (car cmds))
         (setq tc (rappend (TC–tasm–body (car cmds)) tc))
         (setq cmds (cdr cmds)))
        ((and (symbolp (car cmds)) (integerp (car (cdr cmds))))
         (setq tc
               (rappend
                (TC–prefix (TC–assemble–op (car cmds)) (car (cdr cmds)))
                tc))
         (setq cmds (cdr (cdr cmds))))
        (T
         (errorstop 39))))
    (reverse tc)))
```

Comments:

The implementation handles the nesting of instruction sequences as specified by the ...–notation.

Name: tc-prefix.definition

Part of: TASM to TC–compiler

Specification:

$$prefix(opr, e) := \begin{cases} opr(e) & \text{, if } 0 \leq e < 16 \\ prefix(\texttt{pfix}, e \gg 4) \cdot opr(e \wedge \#x0F) & \text{, if } e \geq 16 \\ prefix(\texttt{nfix}, \overline{e} \gg 4) \cdot opr(e \wedge \#x0F) & \text{, if } e < 0 \end{cases}$$

Source–Code:

```
(defun TC–prefix (op e)
  (cond
    ((< e 0) (append (TC–prefix #x6 (floor (– –1  e) 16))
                     (list (+ (* 16 op) (mod e 16)))))
    ((>= e 16) (append (TC–prefix #x2 (floor e 16))
                       (list (+ (* 16 op) (mod e 16)))))
    (T (list (+ (* 16 op) e)))))
```

Issues:

Calculate the `pfix`/`nfix`–chain for a given operator/operand pair.

Name: tc-assemble-op.definition

Part of: TASM to TC–compiler

Specification:

Direct Operation Codes

0x	0000	j	jump
1x	0001	ldlp	load local pointer
2x	0010	pfix	prefix
3x	0011	ldnl	load non local
4x	0100	ldc	load constant
5x	0101	ldnlp	load non local pointer
6x	0110	nfix	negative prefix
7x	0111	ldl	load local
8x	1000	adc	add constant
9x	1001	call	call
Ax	1010	cj	conditional jump
Bx	1011	ajw	adjust workspace
Cx	1100	eqc	equal to constant
Dx	1101	stl	store local
Ex	1110	stnl	store non local
Fx	1111	opr	operate

Source–Code:

```
(defun TC–assemble–op (mnemonic)
  (cond
    ((eql mnemonic 'adc)   #x8)
    ((eql mnemonic 'ajw)   #xB)
    ((eql mnemonic 'call)  #x9)
    ((eql mnemonic 'cj)    #xA)
    ((eql mnemonic 'eqc)   #xC)
    ((eql mnemonic 'j)     #x0)
    ((eql mnemonic 'ldc)   #x4)
    ((eql mnemonic 'ldl)   #x7)
    ((eql mnemonic 'ldlp)  #x1)
    ((eql mnemonic 'ldnl)  #x3)
    ((eql mnemonic 'ldnlp) #x5)
    ((eql mnemonic 'nfix)  #x6)
    ((eql mnemonic 'opr)   #xF)
    ((eql mnemonic 'pfix)  #x2)
    ((eql mnemonic 'stl)   #xD)
    ((eql mnemonic 'stnl)  #xE)
    (T (errorstop 38))))
```

Bibliography

[CM86] L. Chirica and D. Martin. Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems.* 8(2):185–214, 1986.

[GDG+96] W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Verification: The *Verifix* Approach. In: P. Fritzson (Ed.): *Proceedings of the Poster Session of CC '96 – International Conference on Compiler Construction.* IDA Technical Report LiTH-IDA-R-96-12, Linkøping, Sweden, 1996.

[GGH+97] T. Gaul, G. Goos, A. Heberle, W. Zimmermann, and W. Goerigk. An Architecture for Verified Compiler Construction. In: *Joint Modular Languages Conference JMLC'97.* Linz, Austria, 1997.

[GH96] W. Goerigk and U. Hoffmann. The Compiler Implementation Language ComLisp. Technical Report Verifix/CAU/1.7, CAU Kiel, 1996.

[GH98a] W. Goerigk and U. Hoffmann. Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. In: *Proceedings FM-TRENDS'98 International Workshop on Current Trends in Applied Formal Methods*, Lecture Notes in Computer Science. Springer Verlag, Boppard, 1998. To appear.

[GH98b] W. Goerigk and U. Hoffmann. The Compiling Specification from ComLisp to Executable Machine Code. Technical Report Nr. 9713, Institut für Informatik, CAU, Kiel, 1998. In Preparation.

[GMO96] W. Goerigk and M. Müller-Olm. Erhaltung partieller Korrektheit bei beschränkten Maschinenressourcen. – Eine Beweisskizze –. Technical Report Verifix/CAU/2.5, CAU Kiel, 1996.

[Hof96] U. Hoffmann. Über die korrekte Implementierung von Compilern. In: *Workshop "Alternative Konzepte für Sprachen und Rechner"*, pages 94–105. Bad Honnef, 1996. Also available as Technical Report Verifix/CAU/3.1.

[Hof97a] U. Hoffmann. Correct Implementation of Compiler Programs. In: R. Berghammer and F. Simon (Eds.): *Workshop on Programming Languages and Fundamentals of Programming, Avendorf September 1997*, Technical Report 9717, pages 127–138. Institut für Informatik, CAU, Kiel, 1997.

[Hof97b] U. Hoffmann. Korrekte Implementierung von Übersetzungsspezifikationen in hoher Programmiersprache. In: H. Kuchen (Ed.): *Arbeitstagung Programmiersprachen.*

Arbeitsbericht des Institutes für Wirtschaftsinformatik 58, pages 13–20. Westfälische Wilhelms-Universität Münster, 1997.

[Hof98a] U. Hoffmann. *Compiler Implementation Verification through Rigorous Syntactical Code Inspection.* PhD thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, Kiel, 1998. In Preparation.

[Hof98b] U. Hoffmann. Towards a Trusted Tool-Chain for Software Development. In: R. Berghammer and Y. Lakhneche (Eds.): *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science. Springer-Verlag, Malente (Germany), 1998. Submitted for Publication.

[Lan97a] H. Langmaack. Contribution to Goodenough's and Gerhart's Theory of Software Testing and Verification: Relation between Strong Compiler Test and Compiler Implementation Verification. In: C. Freksa, M. Jantzen, and R. Valk (Eds.). *Foundations of Computer Science: Potential-Theory-Cognition. LNCS.* 1337:321–335, Springer Verlag, Berlin, Heidelberg, New York, 1997.

[Lan97b] H. Langmaack. Softwareengineering zur Zertifizierung von Systemen: Spezifikations-, Implementierungs-, Übersetzerkorrektheit. *Informationstechnik und Technische Informatik it-ti.* 97(3):41–47, Oldenbourg Verlag, 1997.

[Lan97c] H. Langmaack. Theoretische Informatik ist Grundlage für das sichere Beherrschen realistischer Software und Systeme. In: K. Brunnstein and H. Oberquelle (Eds.). *25 Jahre Informatik an der Universität Hamburg. Informatik: Stand, Trends, Visionen.* pages 47–62, 1997.

[Mic98] D. Michelsen. Korrektheit der Daten- und Operationsverfeinerung für eine applikative Sprache mit automatischer Speicherbereinigung. Master's thesis, Institut für Informatik, CAU, Kiel, 1998.

[MO96] M. Müller-Olm. Three Views on Preservation of Partial Correctness. Technical Report Verifix/CAU/5.1, CAU Kiel, 1996.

[MO97] M. Müller-Olm. *Modular Compiler Verification*, *Lecture Notes in Computer Science* 1283. Springer-Verlag, Berlin, Heidelberg, New York, 1997.

[Moo88] J Moore. Piton: A verified assembly level language. Technical Report 22, Comp. Logic Inc, Austin, Texas, 1988.

[Moo96] J Moore. *Piton, A Mechanically Verified Assembly-Level Language.* Kluwer Academic Publishers, 1996.

[Par90] H. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[Tho84] K. Thompson. Reflections on Trusting Trust. *Communications of the ACM.* 27(8):761–763, 1984. Also in ACM Turing Award Lectures: The First Twenty Years 1965-1985, ACM Press, 1987, and in Computers Under Attack: Intruders, Worms, and Viruses Copyright, ACM Press 1990.

Appendix A

Runtime System

The runtimesystem is a collection of standard ComLisp–definition. They are considered to be part of every compiler program. The two major parts of the runtimesystem define input and output routines for s–expressions, namely the functions `PRINT` and `READ`.

A.1 Runtime System: Code Review

Name: runtimesystem.definition

Source–Code:

```
;;;-----------------------------------------------------------------------
;;; Verifix: Proven correct compilers
;;; Copyright (C) 1995 Wolfgang Goerigk, Ulrich Hoffmann
;;; Christian–Albrechts–Universitaet zu Kiel, Germany
;;;-----------------------------------------------------------------------
;;; Function : The ComLisp Runtimesystem (Lisp Part)
;;; Syntax:    ComLisp
;;;
;;; $Revision: 1.56 $
;;; $Id: runtimesystem.comlisp,v 1.56 1998/09/24 07:21:28 uho Exp $
;;;-----------------------------------------------------------------------


;;; Control–Variable for Conditional Printing
(defvar *print*)  (setq *print* NIL)

;;-----------------------------------------------------------------------
;; Conditional Printing
;;-----------------------------------------------------------------------
(defun cp (x)
  (if *print* (print x))
  x)

;;;-----------------------------------------------------------------------
;;; Printer
;;;-----------------------------------------------------------------------
(defun terpri ()
  (write–char (code–char 10))
  NIL)

(defun print (d)
  (terpri) (prin1 d) d)

(defun prin1 (d)
  (cond
    ((consp d)     (write–char #\() (prin1 (car d)) (printrest (cdr d)))
    ((null d)      (write–char #\() (write–char #\)))
    ((symbolp d)   (print–symbol d))
    ((integerp d)  (print–integer d))
    ((characterp d)(print–character d))
    ((stringp d)   (write–char #\") (print–string d) (write–char #\"))
    (T (errorstop 3)))
  d)


(defun printrest (d)
  (print–elements d)
  (write–char #\))

(defun print–elements (d)
```

```
  (do ()
      ((null (consp d)))
    (write-char #\space) (prin1 (car d))
    (setq d (cdr d)))
  (if d
      (progn
        (write-char #\space) (write-char #\.) (write-char #\space)
        (prin1 d))))

(defun print-symbol (d)
  (print-string (symbol-name d)))

(defun print-string (name)
  (let* ((len (length name)) (i 0))
    (do ()
        ((eql len 0))
      (let ((c (aref name i)))
        (if (or (eql c #\") (eql c #\\))
            (write-char #\\))
        (write-char c))
      (setq i (+ i 1))
      (setq len (- len 1)))))

(defvar *print-base*) (setq *print-base* 10)

(defun print-integer (n)
  (cond
   ((< n 0) (write-char #\-) (print-integer (- 0 n)))
   (T
    (let ((r (floor n *print-base*))
      (digit (mod n *print-base*)))
      (if (null (eql r 0)) (print-integer r))
      (if (>= digit 10)
      (write-char (code-char (+ (- digit 10) (char-code #\A))))
    (write-char (code-char (+ digit (char-code #\0))))))))
  n)

(defun print-character (d)
  (write-char #\#) (write-char #\\)
  (cond
    ((< 32 (char-code d)) (write-char d))
    ((eql d #\newline)
     (write-char #\n) (write-char #\e) (write-char #\w)
     (write-char #\l) (write-char #\i) (write-char #\n) (write-char #\e))
    ((eql d #\tab)
     (write-char #\t) (write-char #\a) (write-char #\b))
    ((eql d #\space)
     (write-char #\s) (write-char #\p) (write-char #\a)
     (write-char #\c) (write-char #\e))
    (T (write-char #\^)
       (write-char (code-char (+ (- (char-code d) 1) (char-code #\a)))))))
```

;;---

```
;; print-list prints a list without surrounding parenthesis
;;-------------------------------------------------------------------------------
(defun print-list (l)
  (cond
    ((null l))
    (T (print (car l)) (print-list (cdr l)))))

;;-------------------------------------------------------------------------------
;; princ
;;-------------------------------------------------------------------------------
(defun princ (d)
  (cond
    ((consp d)
     (write-char #\() (princ (car d))
     (let ((e (cdr d)))
       (do ()
           ((null (consp e)))
         (write-char #\space) (princ (car e))
         (setq e (cdr e)))
       (if e
           (progn
             (write-char #\space) (write-char #\.) (write-char #\space)
             (princ e) (write-char #\)))
           (write-char #\))))))
    ((null d) (write-char #\() (write-char #\)))
    ((symbolp d) (princ (symbol-name d)))
    ((integerp d) (print-integer d))
    ((characterp d) (write-char d))
    ((stringp d) (let ((n (length d)) (i 0))
                   (do ()
                       ((>= i n))
                     (princ (aref d i))
                     (setq i (+ 1 i)))))
    (T (errorstop 2)))
  d)

(defun write-eof ()
  (write-char #\^z))



;;;-------------------------------------------------------------------------------
;;; Reader
;;;-------------------------------------------------------------------------------
;;;
;;;
;;; Meta symbols:  ::=  |  *
;;; Non terminals:  <ccc>
;;;
;;;
;;; Parser grammer:
;;; ~~~~~~~~~~~~~~~
;;; <sexpr>     ::= <atom> | <list> | QUOTE <sexpr>
```

```
;;; <atom>      ::= STRING | CHARACTER | NUMBER | SYMBOL | LPAREN RPAREN
;;;
;;; <list>      ::= LPAREN <sexpr> <listrest>
;;; <listrest>  ::= DOT <sexpr> RPAREN  |
;;;                 RPAREN |
;;;                 <sexpr> <listrest>
;;;
;;; Scanner grammer:
;;; ~~~~~~~~~~~~~~~~
;;; <tokenseq>  ::= <tokens>* <layout>*
;;;
;;; <tokens>    ::= <layout>* <toks>
;;;
;;; <toks>      ::= <lparen> | <dot> | <rparen> | <quote> |
;;;                 <character> | <symbol> |
;;;                 <number> <lparen> |
;;;                 <number> <rparen> |
;;;                 <number> <layout>
;;;
;;; <lparen>    ::= (                          -> LPAREN
;;; <dot>       ::= .                          -> DOT
;;; <rparen>    ::= )                          -> RPAREN
;;; <quote>     ::= '                          -> QUOTE
;;;
;;; <string>    ::= "<string-char>*"           -> (STRING string)
;;;
;;; <character> ::= #\<char>                   -> (CHARACTER character)
;;;
;;; <number>    ::= <hexnumber> | <decnumber>   -> (NUMBER value)
;;;
;;; <hexnumber> ::= +<hexdigit>* | -<hexdigit>* | <hexdigit><hexdigit>*
;;; <hexdigit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
;;;                 A | B | C | D | E | F | a | b | c | d | e | f
;;;
;;; <decnumber> ::= +<decdigit><decdigit>* |
;;;                 -<decdigit><decdigit>* |
;;;                 <decdigit><decdigit>*
;;; <decdigit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
;;;
;;; <symbol>    ::= <nondigit-constituent> <constituent>*    -> (SYMBOL symbol)
;;;
;;; <layout>    ::= <whitespace> | <comment>
;;; <comment>   ::= ; <comment-char>* <newline> | ; <comment-char>*
;;;
;;;
;;;-------------------------------------------------------------------------
;;;              Character classification  (see CLtL2 p.512)
;;;-------------------------------------------------------------------------
;;;
;;; <constituent>    ::= ! | $ | % | * | + | - | / | : | < |
;;;                      = | > | ? | @ | [ | ] | ^ | _ | { | } | ~ |
;;;                      0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
;;;                      A | B | C | D | E | F | G | H | I | J | K | L | M |
```

```
;;;                     N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
;;;                     a | b | c | d | e | f | g | h | i | j | k | l | m |
;;;                     n | o | p | q | r | s | t | u | v | w | x | y | z
;;;
;;; <nondigit-constituent>
;;;                 ::= ! | $ | % | * | + | - | / | : | < |
;;;                     = | > | ? | @ | [ | ] | ^ | _ | { | } | ~ |
;;;                     A | B | C | D | E | F | G | H | I | J | K | L | M |
;;;                     N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
;;;                     a | b | c | d | e | f | g | h | i | j | k | l | m |
;;;                     n | o | p | q | r | s | t | u | v | w | x | y | z
;;;
;;; <whitespace>    ::= <newline> | <space> | <tab> | <eof>
;;;
;;; <terminating>   ::= ( | ) | " | ' | , | ; | `
;;;
;;; <delimiting>    ::= <terminating> | <whitespace>
;;;
;;; <bar>           ::= |
;;;
;;; <char>          ::= <constituent> | . | # | <terminating> |
;;;                     \ | <bar> | <whitespace>
;;;
;;; <comment-char>  ::= <constituent> | # | . | <terminating> |
;;;                     \ | <bar> | <space> | <tab>
;;;
;;; <string-char>   ::= <constituent> | # | . |
;;;                     ( | ) | ' | , | ; | ` |
;;;                     <bar> | <whitespace> | \<char>
;;;
;;; <newline>
;;; <space>
;;; <tab>
;;; <eof>
;;;
;;; This code assumes the ASCII character set
;;;-----------------------------------------------------------------------------

;;; (princ "; reader lacks proper eof-file handling!") (terpri)

;;-----------------------------------------------------------------------------
;; Character attributes
;;-----------------------------------------------------------------------------
(defun whitespace-p (ch)
  ;; Test if the character CH is a whitespace.
  (>= (char-code #\space) (char-code ch)))

(defun terminating-p (ch)
  ;; Test if the character CH is a terminating character
  (or (eql ch #\()
  (or (eql ch #\))
  (or (eql ch #\")
  (or (eql ch #\')
```

```
  (or (eql ch #\,)
  (or (eql ch #\;)
      (eql ch #\`)))))))))

(defun eof-p ()
  ;; Test if input is exhausted
  (eql (peek-char) #\^Z))

(defun eol-p ()
  ;; Test if input is at end of a line
  (eql (peek-char) (code-char 10)))

(defun delimiting-p  (ch)
  ;; Test if the look ahead character is a delimiting character
  (or (whitespace-p ch) (terminating-p ch)))

(defun digit-p (ch base)
  ;; Test if character CH is a valid digit in the given BASE
  (let ((x (char-code (char-upcase ch))))
    (or
     (and (>= x (char-code #\0)) (>= (char-code #\9) x))
     (and (>= x (char-code #\A)) (>= (- (+ (char-code #\A) base) 11) x)))))




;;-------------------------------------------------------------------------
;; String scanning
;;-------------------------------------------------------------------------

(defun scan-string ()
  ;; Read a string delimited by the '"' character.
  ;; Within the string escaping via the \ character is possible.
  ;; Returns tokenclass STRING and the list of characters read.
  (read-char)               ; skip leading '"'
  (let ((cs NIL))
    (do ()
    ((eql (peek-char) #\"))                     ; trailing '"' ?
    (if (eql (peek-char) #\\) (read-char))
    (if (eof-p) (errorstop 4))
    (setq cs (cons (peek-char) cs))
    (read-char))
    (read-char)
    (cons 'STRING (reverse cs))))



;;-------------------------------------------------------------------------
;; scan-character
;;-------------------------------------------------------------------------

(defun scan-character ()
  ;; Read a character
```

```
  ;; Prefix #\ has already been read
  ;; Handle simple characters #\C
  ;;        control characters #\^C
  ;; and    symbolic characters #\newline #\space and #\tab
  (let ((ch  (peek-char)))
    (read-char)
    (cond
      ((and (eql ch #\^) (null (delimiting-p (peek-char))))
       (setq ch (code-char
                  (- (char-code (char-upcase (read-char))) (char-code #\@)))))

      ((delimiting-p (peek-char)) NIL)

      ((eql (char-upcase ch) #\N)
       (setq ch NIL)
       (if (eql (char-upcase (read-char)) #\E)
       (if (eql (char-upcase (read-char)) #\W)
       (if (eql (char-upcase (read-char)) #\L)
       (if (eql (char-upcase (read-char)) #\I)
       (if (eql (char-upcase (read-char)) #\N)
       (if (eql (char-upcase (read-char)) #\E)
           (setq ch #\newline)))))))
       (if (or (null ch) (null (delimiting-p (peek-char)))) (errorstop 52)))

      ((eql (char-upcase ch) #\T)
       (setq ch NIL)
       (if (eql (char-upcase (read-char)) #\A)
       (if (eql (char-upcase (read-char)) #\B)
           (setq ch #\tab)))
       (if (or (null ch) (null (delimiting-p (peek-char)))) (errorstop 52)))

      ((eql (char-upcase ch) #\S)
       (setq ch NIL)
       (if (eql (char-upcase (read-char)) #\P)
       (if (eql (char-upcase (read-char)) #\A)
       (if (eql (char-upcase (read-char)) #\C)
       (if (eql (char-upcase (read-char)) #\E)
           (setq ch #\space)))))
       (if (or (null ch) (null (delimiting-p (peek-char)))) (errorstop 52)))

      (T
       (errorstop 52)))

    (cons 'CHARACTER ch)))


;;-------------------------------------------------------------------------
;; Number scanning
;;-------------------------------------------------------------------------

(defun scan-hexnumber ()
  ;; Scan a signed hexadecimal number from the input, #X is already read.
  ;; Stop scanning at the first non hexadecimal digit.
```

```
  ;; Return tokenclass HEXNUMBER and the list of characters read.
  (let ((cs NIL))
    (cond
     ((member (peek-char) (list #\- #\+))
      (setq cs (cons (peek-char) cs))
      (read-char))
     ((digit-p (peek-char) 16))
     (T (errorstop 5)))
    (if (eof-p) (errorstop 4))
    (do ()
        ((null (digit-p (peek-char) 16)))
    (setq cs (cons (peek-char) cs))
    (read-char)
    (if (eof-p) (errorstop 4)))
    (if (delimiting-p (peek-char))
        (cons 'HEXNUMBER (reverse cs))
      (errorstop 6))))

(defun scan-decnumber ()
  ;; Scan a signed decimal number from the input.
  ;; Stop scanning at the first terminating character and
  ;; Return tokenclass and the sequence of characters read.
  (let ((cs NIL))
    (cond
     ((member (peek-char) (list #\- #\+))
      (setq cs (cons (peek-char) cs))
      (read-char))
     ((digit-p (peek-char) 10))
     (T (errorstop 7)))
    (if (eof-p) (errorstop 4))
    (cond
     ((delimiting-p (peek-char)) (cons 'SYMBOL cs))
     (T
      (do ()
      ((null (digit-p (peek-char) 10)))
      (setq cs (cons (peek-char) cs))
      (read-char)
      (if (eof-p) (errorstop 4)))
      (if (delimiting-p (peek-char))
      (cons 'DECNUMBER (reverse cs))
    (scan-symbol cs))))))


(defun val (ch)
  ;; Determine the value of the digit CH.
  (let ((v (- (char-code ch) (char-code #\0))))
    (if (< v 10)
        v
        (+ 10 (- (char-code ch) (char-code #\A))))))

(defun value (digits base)
  ;; Determine value of DIGIT Sequence in given BASE
  (let ((v 0)
```

```
        (sign -1))
    (cond
      ((eql (car digits) #\+) (setq digits (cdr digits)) (setq sign -1))
      ((eql (car digits) #\-) (setq digits (cdr digits)) (setq sign 1)))
    (do ()
        ((null digits))
      (setq v (- (* v base) (val (car digits))))
      (setq digits (cdr digits)))
    (* sign v)))


;;-----------------------------------------------------------------------
;; Symbol scanning
;;-----------------------------------------------------------------------

(defun scan-symbol (prefix)
  ;; Scan a print-name from the input.
  ;; Return tokenclass SYMBOL and the characters read.
  (let ((cs (reverse prefix)))
    (do ()
    ((delimiting-p (peek-char)))
    (if (eof-p) (errorstop 4))
    (if (eql (peek-char) #\\)  (errorstop 9)) ; single-esc
    (if (eql (peek-char) #\|)  (errorstop 10)) ; multi-esc
    (if (eql (peek-char) #\.)  (errorstop 11)) ; dot
    (if (eql (peek-char) #\#)  (errorstop 12)) ; hash
    (setq cs (cons (peek-char) cs))
    (read-char))
    (cons 'SYMBOL (reverse cs))))

(defun read-token ()
  ;; Get the next token from the input and classify it into the token classes,
  ;; EOF, RPAREN, LPAREN, QUOTE, STRING, SYMBOL, NUMBER, and CHARACTER.  If
  ;; class is STRING, SYMBOL, NUMBER or CHARACTER, return pair (class . value)
  (do ()
      ((and
        (or (eof-p)
            (null (whitespace-p (peek-char))))
        (null (eql (peek-char) #\;))))
      (if (eql (peek-char) #\;)
      (do ()
          ((or (eof-p) (eol-p)))
          (read-char))
        (read-char)))
  (cond
   ((eof-p)  'EOF)
   ((eql (peek-char) #\))
    (read-char) 'RPAREN)        ; right paren
   ((eql (peek-char) #\()
    (read-char) 'LPAREN)        ; left paren
   ((eql (peek-char) #\')
    (read-char) 'QUOTE)         ; quote
   ((eql (peek-char) #\")
```

```
    (scan-string))          ; string
   ((eql (peek-char) #\#)
    (read-char)
    (cond
     ;;  ((eql (peek-char) #\() (read-char)  'HASHPAREN)
     ((eql (peek-char) #\\)
      (read-char)
      (scan-character))
     ((or (eql (peek-char) #\X) (eql (peek-char) #\x))
      (read-char)
      (scan-hexnumber))         ; hexadecimal num
     ((or (eql (peek-char) #\B) (eql (peek-char) #\b))
      (errorstop 13))           ; binary num
     (T
      (errorstop 14))))
   ((or (eql (peek-char) #\-) (or (eql (peek-char) #\+)
                  (digit-p (peek-char) 10)))
    (scan-decnumber))
   ((eql (peek-char) #\.)
    (read-char)
    (if (delimiting-p (peek-char))
    'DOT
      (errorstop 15)))
   ((or (eql (peek-char) #\,) (eql (peek-char) #\`))
    (errorstop 16))
   (T (scan-symbol NIL))))


(defun get-token ()
  (let ((x (read-token)))
    (if (consp x)
    (let ((class (car x)))
      (cond
       ((eql class 'HEXNUMBER)
        (cons 'NUMBER (value (cdr x) 16)))
       ((eql class 'DECNUMBER)
        (cons 'NUMBER (value (cdr x) 10)))
       ((eql class 'SYMBOL)
        (cons 'SYMBOL (intern (coerce (upper-case (cdr x)) 'STRING))))
       ((eql class 'STRING)
        (cons 'STRING (coerce (cdr x) 'STRING)))
       (T x)))          ; CHARACTER
      x)))              ; LPAREN RPAREN DOT EOF QUOTE



;;-----------------------------------------------------------------------
;; Parser
;;-----------------------------------------------------------------------
(defun listrest ()
  ;; Parse the end of a list.
  (let* ((token (get-token))
         (class (if (symbolp token) token (car token))))
```

```
    (cond
      ((eql class 'EOF) (errorstop 17))
      ((eql class 'RPAREN) 'NIL)
      ((eql class 'DOT)
       (let ((rest (sexpr (get-token))))
         (if (eql (get-token) 'RPAREN)
             rest
             (errorstop 18))))
      (T (cons (sexpr token) (listrest))))))

(defun sexpr (token)
  (let ((class (if (symbolp token) token (car token))))
    (cond
      ((eql class 'EOF) (errorstop 19))
      ;;      ((eql class 'HASHPAREN) (read-array 0))
      ((eql class 'QUOTE)
       (cons 'QUOTE (cons (sexpr (get-token)) NIL)))
      ((eql class 'LPAREN)
       (let ((token (get-token)))
         (if (eql token 'RPAREN)
             NIL
             (cons (sexpr token) (listrest)))))
      ((eql class 'DOT) (errorstop 20))
      (T (cdr token)))))                ; STRING SYMBOL NUMBER CHARACTER

(defun read ()
  (let ((token (get-token)))
    (if (eql token 'EOF)
        #\^Z
        (sexpr token))))

;;-------------------------------------------------------------------------------
;; Read and Print sequences
;;-------------------------------------------------------------------------------
(defun read-sequence ()
  (let ((l nil) (s (read)))
    (do () ((eql s #\^z))
      (setq l (cons s l))
      (setq s (read)))
    (reverse l)))

(defun print-sequence (p)
  (do () ((null p))
    (print (car p))
    (setq p (cdr p)))
  p)

(defun print-hexadecimal (p)
  (let ((old-print-base *print-base*))
    (setq *print-base* 16)
    (print p)
    (setq *print-base* old-print-base)))
```

```
;;;-------------------------------------------------------------------------
;;; Additional Functions
;;;-------------------------------------------------------------------------

(defun errorstop (n)
  (princ #\E) (princ #\r) (princ #\r) (prin1 n) (terpri) (abort))

(defun append (l1 l2)
  (if (null l1)
      l2
      (cons
       (car l1)
       (append (cdr l1) l2))))

(defun reverse (l)
  (let ((r NIL))
    (do ()
     ((null l))
     (setq r (cons (car l) r))
     (setq l (cdr l)))
    r))

(defun rappend (l1 l2)
  (do ()
      ((null l1))
    (setq l2 (cons (car l1) l2))
    (setq l1 (cdr l1)))
  l2)

(defun assoc (item alist)
  (cond ((null alist) nil)
        ((eql item (car (car alist))) (car alist))
        (t (assoc item (cdr alist)))))

(defun equal (s1 s2)
  (cond
    ((consp s1)
     (if (consp s2)
         (and (equal (car s1) (car s2))
              (equal (cdr s1) (cdr s2)))
         NIL))
    ((stringp s1)
     (if (and (stringp s2) (eql (length s1) (length s2)))
         (let ((i (- (length s1) 1)))
           (do ()
               ((or (< i 0) (null (eql (aref s1 i) (aref s2 i)))))
             (setq i (- i 1)))
           (< i 0))
         NIL))
    ((eql s1 s2))))
```

```
(defun assoc-equal (item alist)
  (cond ((null alist) nil)
        ((equal item (car (car alist))) (car alist))
        (t (assoc-equal item (cdr alist)))))

(defun member (el li)
  (cond
    ((null li) nil)
    ((eql el (car li)) li)
    (T (member el (cdr li)))))

(defun char-upcase (c)
  (let ((cc (char-code c)))
    (if (and (>= cc (char-code #\a)) (>= (char-code #\z) cc))
        (code-char (+ (- cc (char-code #\a)) (char-code #\A)))
        c)))

(defun upper-case (l)
  (if (null l)
      NIL
    (cons (char-upcase (car l)) (upper-case (cdr l)))))

(defun list-length (l)
  (let ((len 0))
    (do ()
        ((null l))
      (setq len (+ 1 len))
      (setq l (cdr l)))
    len))


;;-------------------------------------------------------------------------------
;; max calculates maximum of two numbers
;;-------------------------------------------------------------------------------
(defun max (x y)  (if (>= x y) x y))

;;-------------------------------------------------------------------------------
;; Functional queues
;;-------------------------------------------------------------------------------
(defun empty-queue () NIL)

(defun put-queue (e q)  (cons e q))

(defun put-queue-list (l q)
  (do ()
      ((null l))
    (setq q (cons (car l) q))
    (setq l (cdr l)))
  q)

(defun queue2list (q) (reverse q))

(defun queue-length (q) (list-length q))
```

```
;;-------------------------------------------------------------------------------
;; Selectors
;;-------------------------------------------------------------------------------
(defun caar (x) (car (car x)))
(defun cadr (x) (car (cdr x)))
(defun cdar (x) (cdr (car x)))
(defun cddr (x) (cdr (cdr x)))
(defun caaar (x) (car (car (car x))))
(defun caadr (x) (car (car (cdr x))))
(defun cadar (x) (car (cdr (car x))))
(defun caddr (x) (car (cdr (cdr x))))
(defun cdaar (x) (cdr (car (car x))))
(defun cdadr (x) (cdr (car (cdr x))))
(defun cddar (x) (cdr (cdr (car x))))
(defun cdddr (x) (cdr (cdr (cdr x))))
(defun caaddr (x) (car (car (cdr (cdr x)))))
(defun cadadr (x) (car (cdr (car (cdr x)))))
(defun caddar (x) (car (cdr (cdr (car x)))))
(defun cadddr (x) (car (cdr (cdr (cdr x)))))
(defun cdddar (x) (cdr (cdr (cdr (car x)))))
```

Appendix B

Core-Runtimesystem

This chapter shows the core runtime system, i. e. the implementation of the operations on s–expressions by integer operations. The core runtime system is effectively generated by the compiler from SIL to C^{int}.

B.1 Core Runtime System: Code Review

Name: core-runtimesystem.definition

Part of: SIL to C^{int}–compiler

Specification:

The core runtime system is specified in the appendix of [GH98b].

Source–Code:

```
;;;---------------------------------------------------------------------------
;;; Verifix: Proven correct compilers
;;; Copyright (C) 1995-98 Wolfgang Goerigk, Ulrich Hoffmann
;;; Christian-Albrechts-Universitaet zu Kiel, Germany
;;;---------------------------------------------------------------------------
;;; Function : The ComLisp Core-Runtimesystem
;;; Syntax:    Cint
;;;
;;; $Revision: 1.8 $
;;; $Id: core-runtimesystem.cint,v 1.8 1998/07/30 08:48:34 uho Exp $
;;;---------------------------------------------------------------------------
;;;
;;; Core runtime system
;;;
;;; heap/stack are represented as arrays of integers (numbered from zero)
;;; where pairs 0/1, 2/3, 4/5, ... code one atomic Lisp-Datum
;;; with the tag/value components in the even/odd numbered cells;
;;; conses/strings take two/length+1 such consecutive pairs
;;; (a regular heap can never contain pointers 'into' a cons/string)

;;{{{ Garbage Collector

;;;---------------------------------------------------------------------------
;;; The Garbage Collector
;;;---------------------------------------------------------------------------
;;;
;;; called by cons/make-string runtime-functions
;;;

;;{{{ Function _collect-garbage

(defun _COLLECT-GARBAGE (8)
  ;; calculate exact top address of heap:
  ;; exact-heaptop=(heaptop-quotetop)/2 -> local (0,1)
  (_SETLOCAL 3 0)
  (_SETLOCAL (_div (_+ _QUOTETOP _HEAPTOP) 2) 1)

  ;; set heaptop to free part
  (_ALLOCATE (_- (_LOCAL 1) _HEAPTOP))

  ;; prepare loop count -> local(2,3)
  (_SETLOCAL 3 2)
  (_SETLOCAL _STACKTOP 3)
  (DO (PROGN)
   (_= (_LOCAL 3) 0)            ; leave if count = 0
```

```
   (PROGN
    (_SETLOCAL (_- (_LOCAL 3) 2) 3) ; decrement count by 2
    ;; read stack location, every stack entry takes 2 words
    (_SETLOCAL (_STACK (_LOCAL 3)) 4)
    (_SETLOCAL (_STACK (_+ (_LOCAL 3) 1)) 5)
    ;; pass exact-heaptop
    (_SETLOCAL (_LOCAL 0) 6)
    (_SETLOCAL (_LOCAL 1) 7)
    ;; collect data
    (_COLLECT 4)
    ;; store back
    (_SETSTACK (_LOCAL 4) (_LOCAL 3))
    (_SETSTACK (_LOCAL 5) (_+ (_LOCAL 3) 1))))

  ;; copy new heap back
  ;; setup pointer
  (_SETLOCAL 3 2)                       ; 3=<integer-tag>
  (_SETLOCAL _HEAPTOP 3)
  (DO
   (PROGN)
   (_= (_LOCAL 1) (_LOCAL 3))       ; leave, if pointer = exact-heaptop
   (PROGN
    (_SETLOCAL (_- (_LOCAL 3) 2) 3)

    (_SETLOCAL 3 4)         ; 3=<integer-tag>
    (_SETLOCAL (_+ (_- (_LOCAL 3) (_LOCAL 1)) _QUOTETOP) 5)

    (_SETLOCAL (_HEAP (_LOCAL 3)) 6)
    (_SETLOCAL (_HEAP (_+ (_LOCAL 3) 1)) 7)
    (_SETHEAP (_LOCAL 6) (_LOCAL 5))
    (_SETHEAP (_LOCAL 7) (_+ 1 (_LOCAL 5)))))

  ;; adjust heap pointer:  allocate(quotetop+heatop-2*exactheaptop)
  (_SETLOCAL (_+ _quotetop _heaptop) 3)
  (_SETLOCAL (_- (_LOCAL 3) (_2* (_LOCAL 1))) 3)
  (_ALLOCATE (_LOCAL 3)))

;;}}}

;;{{{ Function _collect

(defun _collect (4)
  ;; item, exact-heaptop -> newitem
  ;; check, if stack argument is an integer: no collection
  (IF (_= (_LOCAL 0) 3)         ; 3=<integer-tag>
      (PROGN)
  ;; check, if stack argument is a character: no collection
  (IF (_= (_LOCAL 0) 4)         ; 4=<chracter-tag>
      (PROGN)
  ;; check, if stack argument is NIL: no collection
  (IF (_= (_LOCAL 0) 0)         ; 0=<nil-tag>
      (PROGN)
  ;; check, if stack argument is T: no collection
```

```
  (IF (_= (_LOCAL 0) 1)         ; 1=<t-tag>
      (PROGN)
  ;; check, if data is constant (address below quotetop): no collection
  (IF (_< (_LOCAL 1) _QUOTETOP)
      (PROGN)
  ;; check, if already collected: retrieve new heap address from old heap
  (IF (_= (_HEAP (_LOCAL 1)) 7)     ; 7=<collected-tag>
      (PROGN
       (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 1)) 1))
  ;; check, if stack argument is a string: collect-string
  (IF (_= (_LOCAL 0) 6)         ; 6=<string-tag>
      (PROGN
       (_collect-string 0))
  ;; check, if stack argument is a cons: collect-cons
  (IF (_= (_LOCAL 0) 5)         ; 5=<cons-tag>
      (PROGN
       (_collect-cons 0))
  ;; check, if stack argument is a symbol: collect-cons (sic)
  (IF (_= (_LOCAL 0) 2)         ; 2=<symbol-tag>
      (PROGN
       (_collect-cons 0))
  ;; else abort with error 50: gc-error
  (PROGN
   (_SETLOCAL 4 0)          ; 4=<character-tag>
   (_SETLOCAL 53 1)         ; 53=#\5
   (_SETLOCAL 4 2)          ; 4=<character-tag>
   (_SETLOCAL 48 3)         ; 48=#\0
   (_err 0))))))))))))

;;}}}

;;{{{ Function _collect-string

(defun _collect-string (24)
  ;; item, exact-heaptop -> newitem
  ;; calculate new position
  (_SETLOCAL 3 4)                       ; 3=<integer-tag>
  (_SETLOCAL _HEAPTOP 5)
  ;; calculate new item
  (_SETLOCAL (_LOCAL 0) 6)
  (_SETLOCAL (_+ (_LOCAL 5) (_- _QUOTETOP (_LOCAL 3))) 7)
  ;; retrieve size
  (_SETLOCAL (_HEAP (_LOCAL 1)) 8)
  (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 1)) 9)

  ;; allocate space in new heap
  (_ALLOCATE (_* (_+ (_LOCAL 9) 1) 2))
  ;; store forward pointer to old space
  ;; size is overwritten, but a copy is held in local(8,9)
  (_SETHEAP 7 (_LOCAL 1))               ; 7=<collected-tag>
  (_SETHEAP (_LOCAL 7) (_+ (_LOCAL 1) 1)) ; store new position

  ;; copy oldspace to newspace
```

```
  ;; store size to newspace
  (_SETHEAP (_LOCAL 8) (_LOCAL 5))
  (_SETHEAP (_LOCAL 9) (_+ (_LOCAL 5) 1))
  ;; copy string elements,
  ;; count:local(8,9), new-position:local(4,5), old-position:local(0,1)
  (DO (PROGN)
      (_= (_LOCAL 9) 0)
    (PROGN
      (_SETLOCAL (_- (_LOCAL 9) 1) 9)
      (_SETLOCAL (_+ (_LOCAL 1) 2) 1)
      (_SETLOCAL (_+ (_LOCAL 5) 2) 5)
      (_SETLOCAL (_HEAP (_LOCAL 1)) 10)
      (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 1)) 11)
      (_SETHEAP (_LOCAL 10) (_LOCAL 5))
      (_SETHEAP (_LOCAL 11) (_+ (_LOCAL 5) 1))))
  (_SETLOCAL (_LOCAL 6) 0)
  (_SETLOCAL (_LOCAL 7) 1))

;;}}}

;;{{{ Function _collect-cons

(defun _COLLECT-CONS (16)
  ;; item, exact-heaptop -> newitem
  ;; calculate new position
  (_SETLOCAL 3 4)                       ; 3=<integer-tag>
  (_SETLOCAL _HEAPTOP 5)
  ;; calculate new item
  (_SETLOCAL (_LOCAL 0) 6)
  (_SETLOCAL (_+ (_LOCAL 5) (_- _QUOTETOP (_LOCAL 3))) 7)

  ;; retrieve car
  (_SETLOCAL (_HEAP (_LOCAL 1)) 8)
  (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 1)) 9)

  ;; allocate space in new heap
  (_ALLOCATE 4)
  ;; store forward pointer to real new address in old space
  ;; car is overwritten, but a copy is held in local(8,9)
  (_SETHEAP 7 (_LOCAL 1))               ; 7=<collected-tag>
  (_SETHEAP (_LOCAL 7) (_+ (_LOCAL 1) 1)) ; store newitem

  ;; collect car
  (_SETLOCAL (_LOCAL 8) 10)
  (_SETLOCAL (_LOCAL 9) 11)
  (_SETLOCAL (_LOCAL 2) 12)
  (_SETLOCAL (_LOCAL 3) 13)
  (_COLLECT 10)
  (_SETHEAP (_LOCAL 10) (_LOCAL 5))
  (_SETHEAP (_LOCAL 11) (_+ (_LOCAL 5) 1))

  ;; item:local(0,1), exact-heaptop:local(2,3)
  ;; newposition:local(4,5), newitem:local(6,7)
```

```
(DO
 (PROGN
   ;; go to cdr
   (_SETLOCAL (_HEAP (_+ 2 (_LOCAL 1))) 0)
   (_SETLOCAL (_HEAP (_+ 3 (_LOCAL 1))) 1)
   ;; set flag
   (_SETLOCAL 1 8)
   (IF (_!= (_LOCAL 0) 5)             ; 5=<cons-tag>
       (PROGN)
   (IF (_= (_HEAP (_LOCAL 1)) 7)      ; 7=<collected-tag>
       (PROGN)
   (IF (_< (_LOCAL 1) _QUOTETOP)
       (PROGN)
       (_SETLOCAL 0 8)))))
 (_!= (_LOCAL 8) 0)
 (PROGN
   ;; calculate new item
   (_SETLOCAL (_LOCAL 0) 8)
   (_SETLOCAL (_+ _HEAPTOP (_- _QUOTETOP (_LOCAL 3))) 9)

   ;; set cdr pointer in old cell
   (_SETHEAP (_LOCAL 8) (_+ (_LOCAL 5) 2))
   (_SETHEAP (_LOCAL 9) (_+ (_LOCAL 5) 3))

   ;; set new position
   (_SETLOCAL 3 4)            ; 3=<integer-tag>
   (_SETLOCAL _HEAPTOP 5)

   ;; retrieve car
   (_SETLOCAL (_HEAP (_LOCAL 1)) 10)
   (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 1)) 11)

   ;; allocate space in new heap
   (_ALLOCATE 4)
   ;; store forward pointer to old space
   ;; car is overwritten, but a copy is held in local(8,9)
   (_SETHEAP 7 (_LOCAL 1))               ; 7=<collected-tag>
   (_SETHEAP (_LOCAL 9) (_+ (_LOCAL 1) 1)) ; store newitem

   ;; collect car
   (_SETLOCAL (_LOCAL 10) 12)
   (_SETLOCAL (_LOCAL 11) 13)
   (_SETLOCAL (_LOCAL 2) 14)
   (_SETLOCAL (_LOCAL 3) 15)
   (_COLLECT 12)
   (_SETHEAP (_LOCAL 12) (_LOCAL 5))
   (_SETHEAP (_LOCAL 13) (_+ (_LOCAL 5) 1))
   ))
;; collect last item
(_SETLOCAL (_LOCAL 0) 8)
(_SETLOCAL (_LOCAL 1) 9)
(_SETLOCAL (_LOCAL 2) 10)
(_SETLOCAL (_LOCAL 3) 11)
```

```
  (_COLLECT 8)
  ;; store into cdr cell
  (_SETHEAP (_LOCAL 8) (_+ (_LOCAL 5) 2))
  (_SETHEAP (_LOCAL 9) (_+ (_LOCAL 5) 3))
  ;; supply result
  (_SETLOCAL (_LOCAL 6) 0)
  (_SETLOCAL (_LOCAL 7) 1)
)

;;}}}

;;}}}

;;{{{ Primitives

(defun abort (0)
  _ABORT)

(defun _err (6)
  ;; write Err
  (_SETLOCAL 4 4)                       ; 4=<character-tag>
  (_SETLOCAL 69 5)                      ; 69=#\E
  (WRITE-CHAR 4)
  (_SETLOCAL 4 4)                       ; 4=<character-tag>
  (_SETLOCAL 114 5)                     ; 114=#\r
  (WRITE-CHAR 4)
  (_SETLOCAL 4 4)                       ; 4=<character-tag>
  (_SETLOCAL 114 5)                     ; 114=#\r
  (WRITE-CHAR 4)
  ;; write first digit
  (_SETLOCAL (_LOCAL 0) 4)
  (_SETLOCAL (_LOCAL 1) 5)
  (WRITE-CHAR 4)
  ;; write second digit
  (_SETLOCAL (_LOCAL 2) 4)
  (_SETLOCAL (_LOCAL 3) 5)
  (WRITE-CHAR 4)
  ;; irregularely terminate
  _ABORT)

(defun _true (2)
  (_SETLOCAL 1 0)                       ; 1=<t-tag>
  (_SETLOCAL 1 1))                      ; 1=<t-value>

(defun _false (2)
  (_SETLOCAL 0 0)                       ; 0=<nil-tag>
  (_SETLOCAL 0 1))                      ; 0=<nil-value>

(defun _truth (1)
  (IF (_LOCAL 0)
      (_true 0)
      (_false 0)))
```

```
;;}}}

;;{{{ Type Assertions

(defun _assert-char (4)
  ;; check for <character-tag>
  (IF (_!= (_LOCAL 0) 4)                ; 4=<character-tag>
      (PROGN                ; error 43: crts: character expected
        (_SETLOCAL  4 0)                ; 4=<character-tag>
        (_SETLOCAL 52 1)                ; #\4
        (_SETLOCAL  4 2)                ; 4=<character-tag>
        (_SETLOCAL 51 3)                ; #\3
        (_err 0))))

(defun _assert-cons (4)
  ;; check for <cons-tag>
  (IF (_!= (_LOCAL 0) 5)        ; 5=<cons-tag>
      (PROGN                ; error 44: crts: cons expected
        (_SETLOCAL  4 0)                ; 4=<character-tag>
        (_SETLOCAL 52 1)                ; #\4
        (_SETLOCAL  4 2)                ; 4=<character-tag>
        (_SETLOCAL 52 3)                ; #\4
        (_err 0))))

(defun _assert-integer (4)
  ;; check for <cons-tag>
  (IF (_!= (_LOCAL 0) 3)        ; 3=<integer-tag>
      (PROGN                ; error 40: crts: integer expected
        (_SETLOCAL  4 0)                ; 4=<character-tag>
        (_SETLOCAL 52 1)                ; #\4
        (_SETLOCAL  4 2)                ; 4=<character-tag>
        (_SETLOCAL 48 3)                ; #\0
        (_err 0))))

(defun _assert-string (4)
  ;; check for <string-tag>
  (IF (_!= (_LOCAL 0) 6)        ; 6=<integer-tag>
      (PROGN                ; error 49: crts: string expected
        (_SETLOCAL  4 0)                ; 4=<character-tag>
        (_SETLOCAL 52 1)                ; #\4
        (_SETLOCAL  4 2)                ; 4=<character-tag>
        (_SETLOCAL 57 3)                ; #\9
        (_err 0))))

;;}}}

;;{{{ Errors

(defun _eql-error (4)
  ;; raise error 41: crts: eql error
  (_SETLOCAL  4 0)                      ; 4=<character-tag>
  (_SETLOCAL 52 1)                      ; #\4
  (_SETLOCAL  4 2)                      ; 4=<character-tag>
```

```
  (_SETLOCAL 49 3)                      ; #\1
  (_err 0))

(defun _cc-error (4)
  ;; raise error 42: crts: argument out of bounds in code-char
  (_SETLOCAL  4 0)                      ; 4=<character-tag>
  (_SETLOCAL 52 1)                      ; #\4
  (_SETLOCAL  4 2)                      ; 4=<character-tag>
  (_SETLOCAL 50 3)                      ; #\2
  (_err 0))

(defun _aref-error (4)
  ;; raise error 45: crts: error in aref expression
  (_SETLOCAL  4 0)                      ; 4=<character-tag>
  (_SETLOCAL 52 1)                      ; #\4
  (_SETLOCAL  4 2)                      ; 4=<character-tag>
  (_SETLOCAL 53 3)                      ; #\5
  (_err 0))

;;}}}

;;{{{ Type predicates

(defun null (1)
  ;; check, that type tag is <nil-tag>
  (_SETLOCAL (_= (_LOCAL 0) 0) 0)   ; 0=<nil-tag>
  ;; form T or NIL
  (_truth 0))

(defun symbolp (1)
  ;; check, that type tag is <nil-tag>, <t-tag> or <symbol-tag>
  (_SETLOCAL (_< (_LOCAL 0) 3) 0)       ; 3=<integer-tag>
  ;; form T or NIL
  (_truth 0))

(defun integerp (1)
  ;; check, that type tag is <integer-tag>
  (_SETLOCAL (_= (_LOCAL 0) 3) 0)       ; 3=<integer-tag>
  ;; form T or NIL
  (_truth 0))

(defun characterp (1)
  ;; check, that type tag is <character-tag>
  (_SETLOCAL (_= (_LOCAL 0) 4) 0)       ; 4=<character-tag>
  ;; form T or NIL
  (_truth 0))

(defun consp (1)
  ;; check, that type tag is <cons-tag>
  (_SETLOCAL (_= (_LOCAL 0) 5) 0)       ; 5=<cons-tag>
  ;; form T or NIL
  (_truth 0))
```

```
(defun stringp (1)
  ;; check, that type tag is <string-tag>
  (_SETLOCAL (_= (_LOCAL 0) 6) 0)       ; 6=<string-tag>
  ;; form T or NIL
  (_truth 0))

;;}}}

;;{{{ Comparison

(defun < (4)
  ;; assure, two stack arguments are both of type integer
  (_assert-integer 0)
  (_assert-integer 2)
  ;; perform comparison on value cells
  (_SETLOCAL (_< (_LOCAL 1) (_LOCAL 3)) 0)
  ;; form T or NIL
  (_truth 0))

(defun >= (4)
  ;; assure, two stack arguments are both of type integer
  (_assert-integer 0)
  (_assert-integer 2)
  ;; perform comparison on value cells
  (_SETLOCAL (_>= (_LOCAL 1) (_LOCAL 3)) 0)
  ;; form T or NIL
  (_truth 0))

(defun eql (4)
  ;; check for types of both arguments
  (IF (_= (_LOCAL 0) 5)                 ; 5=<cons-tag>
      (PROGN
        (IF (_= (_LOCAL 2) 5)           ; 5=<cons-tag>
            (_eql-error 0))     ; both cons
        (IF (_= (_LOCAL 2) 6)           ; 6=<string-tag>
            (_eql-error 0))     ; cons and string
        (_false 0))         ; cons and not(string or cons)
      (IF (_= (_LOCAL 0) 6)             ; 6=<string-tag>
          (PROGN
            (IF (_= (_LOCAL 2) 5)       ; 5=<cons-tag>
                (_eql-error 0))     ; string and cons
            (IF (_= (_LOCAL 2) 6)       ; 6=<string-tag>
                (_eql-error 0))     ; string and string
            (_false 0))                 ; string and not(string or cons)
      ;; compare value cells
          (PROGN
            (_SETLOCAL (_= (_LOCAL 0) (_LOCAL 2)) 0)
            (IF (_LOCAL 0)
                (_SETLOCAL (_= (_LOCAL 1) (_LOCAL 3)) 0))
            ;; form T or NIL
            (_truth 0)))))

;;}}}
```

```
;;{{{ Arithmetic

(defun + (4)
  ;; assure, two stack arguments are both of type integer
  (_assert-integer 0)
  (_assert-integer 2)
  ;; perform operation on value cells
  (_SETLOCAL (_+ (_LOCAL 1) (_LOCAL 3)) 1))

(defun - (4)
  ;; assure, two stack arguments are both of type integer
  (_assert-integer 0)
  (_assert-integer 2)
  ;; perform operation on value cells
  (_SETLOCAL (_- (_LOCAL 1) (_LOCAL 3)) 1))

(defun * (4)
  ;; assure, two stack arguments are both of type integer
  (_assert-integer 0)
  (_assert-integer 2)
  ;; perform operation on value cells
  (_SETLOCAL (_* (_LOCAL 1) (_LOCAL 3)) 1))

(defun floor (6)
  ;; assure, two stack arguments are both of type integer
  (_assert-integer 0)
  (_assert-integer 2)
  ;; perform operation on value cells
  ;; _DIV is symmetric division, floor must be floored
  (_SETLOCAL (_DIV (_LOCAL 1) (_LOCAL 3)) 5)
  ;; handle special case, if remainder=0
  (IF (_!= (_REM (_LOCAL 1) (_LOCAL 3)) 0)
      ;; correct if divisor and divident have different sign
      (IF (_< (_LOCAL 1) 0)
          (IF (_>= (_LOCAL 3) 0)
              (_SETLOCAL (_- (_LOCAL 5) 1) 5))
          (IF (_< (_LOCAL 3) 0)
              (_SETLOCAL (_- (_LOCAL 5) 1) 5))))
  (_SETLOCAL (_LOCAL 5) 1))

(defun mod (6)
  ;; assure, two stack arguments are both of type integer
  (_assert-integer 0)
  (_assert-integer 2)
  ;; perform operation on value cells
  (_SETLOCAL (_REM (_LOCAL 1) (_LOCAL 3)) 5)
  ;; _REM is remainder for symmetric division,
  ;; mod must be modulus for floored divison
  ;; handle special case if remainder=0
  (IF (_!= (_LOCAL 5) 0)
      ;; correct if divisor and divident have different sign
      (IF (_< (_LOCAL 1) 0)
```

```
          (IF (_>= (_LOCAL 3) 0)
              (_SETLOCAL (_+ (_LOCAL 5) (_LOCAL 3)) 5))
          (IF (_< (_LOCAL 3) 0)
              (_SETLOCAL (_+ (_LOCAL 5) (_LOCAL 3)) 5))))
  (_SETLOCAL (_LOCAL 5) 1))

;;}}}

;;{{{ Conversion

(defun code-char (2)
  ;; assure, stack argument is of type integer
  (_assert-integer 0)
  ;; assure argument is in range [0,255]
  (IF (_< (_LOCAL 1) 0)    (_cc-error 0))
  (IF (_>= (_LOCAL 1) 256) (_cc-error 0))
  ;; change type tag to <character-tag>
  (_SETLOCAL 4 0))                      ; 4=<character-tag>

(defun char-code (2)
  ;; assure, stack argument is of type character
  (_assert-char 0)
  ;; change type tag to <integer-tag>
  (_SETLOCAL 3 0))                      ; 3=<integer-tag>

;;}}}

;;{{{ Input/Output

(defun read-char (2)
  ;; set type tag to <character-tag>
  (_SETLOCAL 4 0)                       ; 4=<character-tag>
  ;; operate on value cell
  (_READ-CHAR 1))

(defun write-char (2)
  ;; assure, stack argument is of type character
  (_assert-char 0)
  ;; operate on value cell
  (_WRITE-CHAR 1))

(defun peek-char (2)
  ;; set type tag to <character-tag>
  (_SETLOCAL 4 0)           ; 4=<character-tag>
  ;; operate on vlaue cell
  (_PEEK-CHAR 1))

;;}}}

;;{{{ CONS-Cells

(defun cons (6)
  ;; check, if memory is available for new cell
```

```
  ;; calculate next free heap address (_quotetop advances twice as fast see gc)
  (_SETLOCAL (_DIV (_+ _HEAPTOP _QUOTETOP) 2) 5)
  (IF (_UNAVAILABLE 8)
      ;; heap full, collect garbage
      (PROGN
       (_COLLECT-GARBAGE 4)
       (_SETLOCAL (_DIV (_+ _HEAPTOP _QUOTETOP) 2) 5)))
  ;; allocate memory for a cell
  (_ALLOCATE 8)
  ;; fill allocated memory
  ;; first stack argument is stored in car part
  (_SETHEAP (_LOCAL 0) (_LOCAL 5))
  (_SETHEAP (_LOCAL 1) (_+ (_LOCAL 5) 1))
  ;; second stack argument is stored in cdr part
  (_SETHEAP (_LOCAL 2) (_+ (_LOCAL 5) 2))
  (_SETHEAP (_LOCAL 3) (_+ (_LOCAL 5) 3))
  ;; set type tag of result to <cons-tag>
  (_SETLOCAL 5 0)                       ; 5=<cons-tag>
  (_SETLOCAL (_LOCAL 5) 1))

(defun car (2)
  ;; return NIL, if stack argument is NIL, check for type tag <nil-tag>
  (IF (_!= (_LOCAL 0) 0)                ; 0=<nil-tag>
      (PROGN
       ;; assure, stack argument is of type cons.
       (_assert-cons 0)
       ;; read Content of Address Register (CAR :-)
       (_SETLOCAL (_HEAP (_LOCAL 1)) 0)
       (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 1)) 1))))

(defun cdr (2)
  ;; return NIL, if stack argument is NIL, check for type tag <nil-tag>
  (IF (_!= (_LOCAL 0) 0)                ; 0=<nil-tag>
      (PROGN
       ;; assure, stack argument is of type cons
       (_assert-cons 0)
       ;; read Content of Decrement Register (CDR :-)
       (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 2)) 0)
       (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 3)) 1))))

;;}}}

;;{{{ Strings

(defun length (2)
  ;; assure, stack argument is of type string
  (_assert-string 0)
  ;; set type tag of result to <integer-tag>
  (_SETLOCAL 3 0)                     ; <integer-tag>
  ;; read length value, which is stored in second string memory cell
  (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 1)) 1))

(defun aref (4)
```

```
  ;; assure, that stack arguments are of type string and integer resp.
  (_assert-string 0)
  (_assert-integer 2)
  ;; assure, index is within bounds
  (if (_< (_LOCAL 3) 0) (_aref-error 0))
  (if (_>= (_LOCAL 3) (_HEAP (_+ (_LOCAL 1) 1))) (_aref-error 0))
  ;; calculate address of string element
  (_SETLOCAL (_2* (_LOCAL 3)) 3)
  ;; retrieve string element
  (_SETLOCAL (_HEAP (_+ (_+ (_LOCAL 1) (_LOCAL 3)) 2)) 0)
  (_SETLOCAL (_HEAP (_+ (_+ (_LOCAL 1) (_LOCAL 3)) 3)) 1))

(defun coerce (10)
  ;; 0,1: character-list
  ;; list-length
  (_SETLOCAL 3 2)                       ; 3=<integer-tag>
  (_SETLOCAL 0 3)
  (_SETLOCAL (_LOCAL 0) 4)
  (_SETLOCAL (_LOCAL 1) 5)
  ;; determine length of list
  (DO
   (PROGN)
   (_= (_LOCAL 4) 0)                    ; 0=<nil-tag>
   (PROGN
     (_SETLOCAL (_+ 1 (_LOCAL 3)) 3)
     (cdr 4)))
  ;; 0,1: character-list   2,3: length
  ;; allocate memory for string
  (_SETLOCAL (_DIV (_+ _HEAPTOP _QUOTETOP) 2) 5)
  (IF (_UNAVAILABLE (_* (_+ (_LOCAL 3) 1) 4))
      (PROGN
        (_COLLECT-GARBAGE 4)
        (_SETLOCAL (_DIV (_+ _HEAPTOP _QUOTETOP) 2) 5)))
  (_ALLOCATE (_* (_+ (_LOCAL 3) 1) 4))
  (_SETLOCAL 6 4)                        ; 6=<string-tag>
  ;; 0,1: character-list  2,3: length 4,5: string
  (_SETHEAP 3 (_LOCAL 5))              ; 3=<integer-tag>
  (_SETHEAP (_LOCAL 3) (_+ (_LOCAL 5) 1))
  (_SETLOCAL (_LOCAL 4) 2)
  (_SETLOCAL (_LOCAL 5) 3)
  (_SETLOCAL 3 4)                       ; 3=<integer-tag>
  (_SETLOCAL (_+ (_LOCAL 5) 2) 5)
  (_SETLOCAL (_LOCAL 0) 6)
  (_SETLOCAL (_LOCAL 1) 7)
  ;; 0,1: character-list  2,3: string  4,5: pointer  6,7: character-list
  ;; copy characters from list to string
  (DO
   (PROGN)
   (_= (_LOCAL 6) 0)                    ; 0=<nil-tag>
   (PROGN
     (_SETLOCAL (_LOCAL 6) 8)
     (_SETLOCAL (_LOCAL 7) 9)
     (car 8)
```

```
     ;; 0,1: character-list  2,3: length 4: pointer 6,7: character-list 8,9: car
     (_assert-char 8)
     (_SETHEAP (_LOCAL 8)  (_LOCAL 5))
     (_SETLOCAL (_+ (_LOCAL 5) 1) 5)
     (_SETHEAP (_LOCAL 9)  (_LOCAL 5))
     (_SETLOCAL (_+ (_LOCAL 5) 1) 5)
     (cdr 6)))
  ;; 0,1: charcter-list  2,3: string 4,5: pointer
  (_SETLOCAL (_LOCAL 2) 0)
  (_SETLOCAL (_LOCAL 3) 1))

;;}}}

;;{{{ Symbol Management

(defun _Tname (4)
  ;; construct printname of T
  ;; build (#\T)
  (_SETLOCAL 4 0)                       ; 4 = <char-tag>
  (_SETLOCAL 84 1)                      ; 78 = #\T
  (_SETLOCAL 0 2)                       ; 0=<nil-tag>
  (_SETLOCAL 0 3)                       ; 0=<nil-value>
  (CONS 0)
  ;; convert to string
  (_SETLOCAL 0 2)                       ; 0=<nil-tag>
  (_SETLOCAL 0 3)                       ; 0=<nil-value>
  (coerce 0))                           ; coerce ignores second parameter

(defun _NILname (8)
  ;; construct printname of NIL
  ;; build (#\N #\I #\L)
  (_SETLOCAL 4 0)                       ; 4 = <char-tag>
  (_SETLOCAL 78 1)                      ; 78 = #\N
  (_SETLOCAL 4 2)                       ; 4 = <char-tag>
  (_SETLOCAL 73 3)                      ; 73 = #\I
  (_SETLOCAL 4 4)                       ; 4 = <char-tag>
  (_SETLOCAL 76 5)                      ; 76 = #\L
  (_SETLOCAL 0 6)                       ; 0=<nil-tag>
  (_SETLOCAL 0 7)                       ; 0=<nil-value>
  (CONS 4)
  (CONS 2)
  (CONS 0)
  ;; convert to string
  (_SETLOCAL 0 2)                       ; 0=<nil-tag>
  (_SETLOCAL 0 3)                       ; 0=<nil-value>
  (coerce 0))                           ; coerce ignores second parameter

(defun symbol-name (4)
  ;; check, if stack argument is of type symbol
  (IF (_= (_LOCAL 0) 2)          ; 2=<symbol-tag>
      (PROGN
       ;; retrieve print name, which is stored at that location
       (_SETLOCAL (_HEAP (_LOCAL 1)) 0)
```

```
       (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 1)) 1))
      ;; check for special cases
      (IF (_= (_LOCAL 0) 0)     ; 0=<nil-tag>
      (_NILname 0)
      (IF (_= (_LOCAL 0) 1)     ; 1=<t-tag>
          (_Tname 0)
          (PROGN            ; neither symbol, nor NIL, T
                ;; error 48: crts symbol expected
                (_SETLOCAL  4 0)                ; 4=<character-tag>
                (_SETLOCAL 52 1)        ; #\4
                (_SETLOCAL  4 2)                ; 4=<character-tag>
                (_SETLOCAL 56 3)        ; #\8
                (_err 0))))))


(defun _string= (10)
  (_SETLOCAL (_HEAP (_+ (_LOCAL 1) 1)) 4) ; 4: length of string1
  (_SETLOCAL (_HEAP (_+ (_LOCAL 3) 1)) 5) ; 5: length of string2
  (IF (_!= (_LOCAL 4) (_LOCAL 5))       ; lengths differ
      (_false 0)
      (PROGN                            ; lengths equal, 4: length
        (_SETLOCAL (_+ (_LOCAL 1) 3) 5) ; 5: address of character1
        (_SETLOCAL (_+ (_LOCAL 3) 3) 6) ; 6: address of character2
        (DO
         (PROGN
           ;; 7: flag := length!=0 AND character1==character2
           (_SETLOCAL (_LOCAL 4) 7)
           (IF (_LOCAL 7)
               (PROGN
                 (_SETLOCAL (_HEAP (_LOCAL 5)) 8) ; 8: ch1
                 (_SETLOCAL (_HEAP (_LOCAL 6)) 9) ; 9: ch2
                 (_SETLOCAL (_= (_LOCAL 8) (_LOCAL 9)) 7))))
         (_= 0 (_LOCAL 7))
         (PROGN
           (_SETLOCAL (_- (_LOCAL 4) 1) 4) ; decrement length
           (_SETLOCAL (_+ (_LOCAL 5) 2) 5) ; increment pointer
           (_SETLOCAL (_+ (_LOCAL 6) 2) 6))) ; increment pointer
        (IF (_LOCAL 4)                  ; length!=0: mismatch
            (_false 0)
            (_true 0)))))

;;(defun _string-member (el li)
;;  (if (null li)
;;      NIL
;;    (if (_string= el (car li))
;;        li
;;      (_string-member el (cdr li)))))

(defun _string-member (8)
  ;; look for string in list
  (IF (_= (_LOCAL 2) 0)
      (PROGN
       ;; list exhausted: not found
```

```
       (_SETLOCAL 0 4)          ; 0=<nil-tag>
       (_SETLOCAL 0 5))         ; 0=<nil-value>
      (PROGN
       ;; retrieve first character
       (_SETLOCAL (_LOCAL 0) 4) (_SETLOCAL (_LOCAL 1) 5)
       (_SETLOCAL (_LOCAL 2) 6) (_SETLOCAL (_LOCAL 3) 7)
       (CAR 6)
       ;; test for match
       (_STRING= 4)
       (IF (_!= (_LOCAL 4) 0)
       (PROGN
        ;; matching string found: return restlist
        (_SETLOCAL (_LOCAL 2) 4)
        (_SETLOCAL (_LOCAL 3) 5))
       (PROGN
        ;; no match: search rest
        (_SETLOCAL (_LOCAL 0) 4)
        (_SETLOCAL (_LOCAL 1) 5)
        (_SETLOCAL (_LOCAL 2) 6)
        (_SETLOCAL (_LOCAL 3) 7)
        (CDR 6)
        (_string-member 4)))))
  (_SETLOCAL (_LOCAL 4) 0)
  (_SETLOCAL (_LOCAL 5) 1))


;;(defun _copy-string (str)
;;  (let ((i (length str))
;;        (r nil))
;;    (loop
;;        (if (>= 0 i) (return))
;;      (setq i (- i 1))
;;      (setq r (cons (aref str i) r)))
;;    (coerce r 'string)))

(defun _copy-string (10)
  ;; determine length of string
  (_SETLOCAL (_LOCAL 0) 2)
  (_SETLOCAL (_LOCAL 1) 3)
  ;; length will abort, if argument is not of type string
  (LENGTH 2)
  ;; construct a list of characters from string
  ;; initital value of list: ()
  (_SETLOCAL 0 4)           ; 0=<nil-tag>
  (_SETLOCAL 0 5)           ; 0=<nil-value>
  ;; loop thru string from last to first character
  (DO
   (PROGN
    ;; leave loop, if the count is below zero
    (_SETLOCAL 3 6)     ; 3=<integer-tag>
    (_SETLOCAL 0 7)
    (_SETLOCAL (_LOCAL 2) 8)
    (_SETLOCAL (_LOCAL 3) 9)
```

```
    (>= 6))
   (_!= (_LOCAL 6) 0)
   (PROGN
    ;; decrement count
    (_SETLOCAL (_LOCAL 2) 6)
    (_SETLOCAL (_LOCAL 3) 7)
    (_SETLOCAL 3 8)         ; 3=<integer-tag>
    (_SETLOCAL 1 9)
    (- 6)
    (_SETLOCAL (_LOCAL 6) 2)
    (_SETLOCAL (_LOCAL 7) 3)
    ;; prepend character to character list
    (_SETLOCAL (_LOCAL 0) 6)
    (_SETLOCAL (_LOCAL 1) 7)
    (_SETLOCAL (_LOCAL 2) 8)
    (_SETLOCAL (_LOCAL 3) 9)
    ;; retrieve character
    (AREF 6)
    (_SETLOCAL (_LOCAL 4) 8)
    (_SETLOCAL (_LOCAL 5) 9)
    ;: prepend
    (CONS 6)
    (_SETLOCAL (_LOCAL 6) 4)
    (_SETLOCAL (_LOCAL 7) 5)))
  ;; convert character list to string
  (_SETLOCAL (_LOCAL 4) 6)
  (_SETLOCAL (_LOCAL 5) 7)
  (_SETLOCAL 0 8)                       ; 0=<nil-tag>
  (_SETLOCAL 0 9)                       ; 0=<nil-value>
  (coerce 6)                            ; coerce ignores second parameter
  (_SETLOCAL (_LOCAL 6) 0)
  (_SETLOCAL (_LOCAL 7) 1))

(defun intern (6)
  ;; assure, stack argument is of type string
  (_assert-string 0)
  ;; look for string in symbol list
  (_SETLOCAL (_LOCAL 0) 2) (_SETLOCAL (_LOCAL 1) 3)
  (_SETLOCAL (_STACK 0) 4) (_SETLOCAL (_STACK 1) 5)
  (_string-member 2)
  (IF (_!= (_LOCAL 2) 0)
      ;; found string in symbol list
      (_SETLOCAL 2 2)           ; 2=<symbol-tag>
      ;; notfound
      (PROGN
       ;; check for special cases. Is it "NIL"
       (_SETLOCAL (_LOCAL 0) 2)
       (_SETLOCAL (_LOCAL 1) 3)
       (_NILname 4)
       (_STRING= 2)
       (IF (_!= (_LOCAL 2) 0)
       (PROGN
        ;; string is "NIL": return NIL
```

```
        (_SETLOCAL 0 2)
        (_SETLOCAL 0 3))
       (PROGN
        ;; check for "T"
        (_SETLOCAL (_LOCAL 0) 2)
        (_SETLOCAL (_LOCAL 1) 3)
        (_Tname 4)
        (_STRING= 2)
        (IF (_!= (_LOCAL 2) 0)
        (PROGN
         ;; string is "T": return T
         (_SETLOCAL 1 2)
         (_SETLOCAL 1 3))
        (PROGN
         ;; new symbol: add to symbol list.
         (_SETLOCAL (_LOCAL 0) 2)
         (_SETLOCAL (_LOCAL 1) 3)
         ;; make a copy
         (_COPY-STRING 2)
         ;; prepend to symbol list
         (_SETLOCAL (_STACK 0) 4)
         (_SETLOCAL (_STACK 1) 5)
         (CONS 2)
                ;; store extended list
                (_SETSTACK (_LOCAL 2) 0)
                (_SETSTACK (_LOCAL 3) 1)
         ;; set type tag to <symbol-tag>
         (_SETLOCAL 2 2)))))))  ; 2=<symbol-tag>
  (_SETLOCAL (_LOCAL 2) 0)
  (_SETLOCAL (_LOCAL 3) 1))

;;}}}

;;; Local Variables: ***
;;; mode:lisp ***
;;; mode:folding ***
;;; End: ***
```

Index